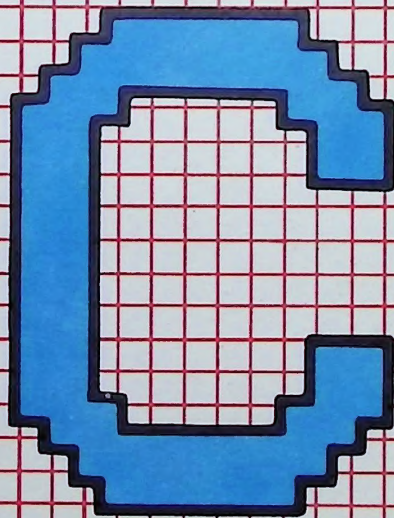


Mikrokomputery

Alex Ragen

LEKSYKON JĘZYKA





Z języka angielskiego przełożyli

Marian Łakomy

Piotr Łakomy

Leksykon
JĘZYKA C

Mikrokomputery

Komitet Redakcyjny

Sekretarz Wojciech Cellary
Zuzanna Grzejszczak
Andrzej Kobus
Przewodniczący Romuald Marczyński
Piotr Misiurewicz
Wojciech Nowakowski
Maciej Stolarski
Halina Tempczyk
Józef Winkowski
Jan Zabrodzki

Alex Ragen

Leksykon JĘZYKA C



Wydawnictwa Naukowo-Techniczne
Warszawa 1990

Dane o oryginale:
Lexicon of C
by
Alex Ragen
© A.Ragen, 1987
First published in 1987 by Sigma Press
98a Water Lane, Wilmslow, SK9 5BB, England

Redaktor *Hanna Włodarska*
Przygotowała do druku *Ewa Eckhardt*
Okładkę i strony tytułowe projektował *Andrzej Pilich*

681.3.06

Książka zawiera opis języka C w formie leksykonowej – jej główna część zawiera uporządkowany alfabetycznie zestaw pojęć i problemów związanych z samym językiem C, jego składnią i zasadami programowania. Ponadto w książce zamieszczono krótki samouczek języka C, który wraz z hasłami z części leksykonowej może stanowić podstawę do opanowania tego języka, zwłaszcza dla tych, którzy znają inne języki wysokiego poziomu. Cenną pomocą będą przy tym rozdziały poświęcone metodom uruchamiania programów i organizacji pracy w zespole programistycznym, a także przykłady programów napisanych zgodnie z zasadami dobrego programowania.
Książka jest przeznaczona dla wszystkich zainteresowanych programowaniem w C, zwłaszcza mających już pewne podstawy tego języka.

Copyright © for the Polish edition
by Wydawnictwa Naukowo-Techniczne
Warszawa 1990

All rights reserved
Printed in Poland

ISBN 83-204-1223-4

Spis treści

1. Wstęp 7
2. Samouczek C 10
3. Uruchamianie programów 21
4. Styl programowania 31
5. Organizacja projektu programowego w języku C 34
6. Leksykon 36
7. Dyrektywy preprocesora 100

Dodatek A. Funkcje standardowe 104
Dodatek B. Funkcje przetwarzania ciągów znaków 135
Dodatek C. Przykładowe programy – szyfrowanie i deszyfrowanie 140
Dodatek D. Tworzenie drzewa binarnego z wyrażenia wielomianowego 146
Dodatek E. Tablica znaków ASCII 152
Skorowidz 155

1. Wstęp

Książka ta powstała w wyniku odczuwanej przez autora potrzeby posiadania użytecznego, obszernego i czytelnego poradnika na temat języka C. Standardowa "biblia" na temat C – książka B. W. Kernighana i D. M. Ritchiego "Język C" (WNT, Warszawa 1987), powszechnie cytowana jako "K&R", wprawdzie zawiera 46 stron szczegółowego przewodnika języka, ale wydaje się, że został on napisany raczej z myślą o teoretykach języków programowania i twórcach kompilatorów niż o programistach. Większość innych książek o języku C – a jest ich wcale niemało – są to samouczki programowania, użyteczne w początkowej fazie nauki, ale mało przydatne później, gdy szuka się szczegółowej odpowiedzi na konkretne pytania.

Ten nieszczęśliwy brak takiego poradnika jest częściowo wynikiem rosnącej popularności C i systemu operacyjnego Unix, z którym język C jest ściśle związany. Wielu producentów sprzętu komputerowego zaadaptowało Unix jako system operacyjny sterujący pracą ich sprzętu, gdyż zwalnia ich to od potrzeby opracowywania własnego systemu, a jednocześnie daje szansę natychmiastowego korzystania z ogromnej biblioteki oprogramowania. Niestety, także ci sami producenci sądzą, że "natychmiastowa dostępność" języka C zwalnia ich od odpowiedzialności za dostarczenie zadowalającej dokumentacji do ich produktów; odsyłają oni użytkownika do standardowej literatury Unixa, chociaż czasami zdarza się, że przedrukowują ją pod własnym znakiem firmowym. Co ciekawsze, bardziej prymitywne języki programowania i własne systemy operacyjne dostarczane przez tradycyjnych producentow dużych komputerów są na ogół dobrze udokumentowane, łącznie ze wszystkimi błędami.

Książka nie jest przeznaczona do czytania jej "od deski do deski"; jej przeznaczeniem jest raczej biurko programisty, gdzie może on do niej zajrzeć w razie potrzeby – w podobny sposób, jak pisarz zagląda w trakcie pracy do słownika. Autor zrobił wszystko, aby hasła w książce były kompletne i nie wymagały odwoływania się do innych, chociaż w większości takie odwołania są zawarte. W celu umieszczenia w całości powiązanych z sobą informacji niezbędne było częściowe powtarzanie materiału.

Wiele podanych przykładów przetestowano przy użyciu kompilatora C z Unixa, kompilatora firmy Microsoft dla systemu MS-DOS i kompilatora Aztec II dla CP/M 80. Wszystkie przykłady powinny działać w podany sposób w tych trzech realizacjach, z wyjątkiem przypadków omówionych w tekście[1].

Podczas pisania tej książki nie istniał zaakceptowany międzynarodowo standard języka C i na czas trwania tego "bezkrólewia" opis podany w części poradnikowej K&R jest uznawany za standard. Amerykański Krajowy Instytut Standardów (ANSI) opracował propozycję standardu języka C, znaną pod nazwą X3J11, ale zapewne przejdzie on jeszcze sporo zmian kosmetycznych nim stanie się kompletny. Pojawił się również konkurencyjny standard X/OPEN. W każdym razie sukces standardu zależy od producentów i ich poparcia dla niego. Wydaje się, że w najbliższym czasie język C będzie nadal dostępny w wielu lekko różniących się wersjach.

Mimo to opis zawarty w książce K&R stanowi podstawę do większości realizacji kompilatorów języka i tutaj także został użyty. Jeśli jakaś cecha jest specyficzna dla konkretnej realizacji, to jest to powiedziane w tekście eksplicite.

Książka rozpoczyna się od krótkiego samouczka C, przeznaczonego dla doświadczonych programistów, którzy potrzebują przypomnieć sobie język w trakcie pracy. Dla programistów dobrze znających inne strukturalne języki programowania, takie jak Pascal lub Algol, samouczek, leksykon, dobry kompilator C i trochę wytrwałości powinno wystarczyć aby byli w stanie osiągnąć znajomość języka C dostateczną do posługiwania się nim. Dobra znajomość przychodzi tylko z doświadczeniem – nie ma bezpłatnych dróg do C.

W dodatkach zawarto przykłady właściwości C opisanych w leksykonie. Programiści powinni przyjrzeć się tym przykładom nie tyle w celu wyciągnięcia z nich bezpośrednich korzyści (chociaż niektóre z nich są całkiem interesujące), ile raczej w celu zapoznania się z tym, jak wyglądają "dobrze napisane programy w C", opracowane przez doświadczonych programistów.

Rozdział na temat uruchamiania programów - tej niewdzięcznej i często frustrującej części pracy programisty – powinien być pomocny zarówno dla początkujących, jak i dla doświadczonych programistów.

Jeszcze parę słów na temat stylu. Autor konsekwetnie pisze programista, ale ma tu na myśli zarówno mężczyzn, jak i kobiety, w przekonaniu, że wszelka próba wyróżniania płci musi być uznana za sztuczną. Autor przeprasza tych, którzy poczują się dotknięci takim podejściem. Na pewno nie zamierzał przez to powiedzieć, że programiści jednej płci bardziej potrzebują tej książki niż programiści płci przeciwnej.

Serdeczne podziękowania należą się Pani Devorah Goldberg, która cierpliwie przeczytała wszystkie wersje książki, poprawiała błędy i sugerowała ulepszenia. Świadomość, że cały tekst będzie poddany jej dogłębnej krytyce, była dla Autora dopingiem do jeszcze większej dokładności. Wszystko co jest dobre w tej książce to

---

[1] Wszystkie programy zawarte w książce przetestowano przy użyciu kompilatorów Turbo C, wersja 1.5 firmy Borland oraz Quick C, wersja 1.01 firmy Microsoft. Niektóre z nich wymagały niewielkich przeróbek i w takim przypadku zamieściliśmy w tekście wersję już dostosowaną do tych kompilatorów. – Przyp. tłum.

zasługa Pani Devorah Goldberg. Za wszystkie błędy, które pozostały mimo jej pracy, odpowiada tylko Autor.

Autor dziękuje także swojej żonie Naomi, która zachęciła go do podjęcia tej pracy i podtrzymywała na duchu przez całe miesiące, stracone urlopy i nie przespane noce.

Jerozolima, maj 1987

2. Samouczek C

Ten samouczek jest przeznaczony dla programistów mających pewne doświadczenie w programowaniu w językach wysokiego poziomu. Znajomość języków strukturalnych, takich jak Pascal, jest szczególnie cenna.

Samouczek jest przeznaczony do czytania razem z hasłami w części leksykalnej książki. Podane tu wyjaśnienia mają na celu raczej wprowadzanie tematów niż ich wyczerpanie; szczegółowe wyjaśnienie zagadnienia wymaga zajrzenia do leksykonu. Doświadczony programista może, przy pewnym wysiłku, nauczyć się języka C korzystając z tego samouczka i leksykonu – pod warunkiem, że ma dostęp do kompilatora C i może poćwiczyć na kilku przykładowych programach.

Język C został opracowany we wczesnych latach siedemdziesiątych przez Dennisa Ritchiego z Bell Laboratory firmy ATT; firma ta brała także udział w opracowaniu systemu operacyjnego Unix. Nazwa sugeruje kontynuację języka B, który powstał w wyniku doświadczeń nad językiem BCPL (ang. *Basic Combined Programming Language*).

Opracowanie języka C nie zostało zainicjowane przez żadną agencję rządową ani producenta komputerów, ale nastąpiło w wyniku potrzeb programisty. Ma on więc te cechy, których można oczekiwać od języka opracowanego dla siebie: jest prosty, ale dostatecznie potężny, zwięzły i precyzyjny oraz słabo stypizowany. Ponadto reguły nie są ściśle przestrzegane.

Nieuniknione jest wrażenie, że jednym z celów jego projektu była minimalizacja liczby uderzeń w klawiaturę, potrzebnych do napisania programu. Liczba instrukcji języka C jest niewielka. Mimo, że istnieje 45 operatorów, jednak większość z nich jest odmianą podstawowych operatorów dostępnych we wszystkich językach. Większość cech dostępnych w innych językach jako ich wbudowane instrukcje, łącznie z wejściem i wyjściem, nie występuje w C i muszą być dostarczone w ramach realizacji języka jako część zewnętrznej biblioteki funkcji. Jednak, mimo swej spartańskiej składni, język C równie dobrze nadaje się do skomplikowanych prac, takich jak pisanie kompilatorów i obliczenia matematyczne, jak i do prac na niskim poziomie, takich jak

operacje na bitach i pisanie programów sterowania urządzeń. Około 95% systemu operacyjnego Unix zostało napisane w C.

Rosnąca popularność C wynika w znacznej mierze z popularności Unixa, chociaż coraz powszechniej są dostępne realizacje C dla wielu systemów operacyjnych i komputerów, od mikrokomputerów poczynając, a na dużych komputerach kończąc. Na przykład dla IBM PC jest dostępnych ponad 20 kompilatorów C. Pisanie programów w C zapewnia ich dość proste przenoszenie z komputera na komputer, gdyż wiele trudnych problemów rozwiązali twórcy kompilatorów, przygotowujący biblioteki funkcji standardowych.

Programiści lubią programować w C, gdyż uważają, że jest łatwy w użyciu; składnia nie przeszkadza w naturalnym wyrażaniu algorytmu. Jednak ta swoboda robienia prawie wszystkiego oznacza, że na programistę spada większa odpowiedzialność za swój produkt. Inną stroną tej "wolności i łatwości użycia" jest "niebezpieczeństwo nadużycia" – język C dopuszcza, ale nie przebacza niedbalstwa.

Mimo, że zmienne mają dobrze określone typy, jednak łatwo jest ominąć nakładane przez język ograniczenia co do mieszania typów. W wyrażeniach arytmetycznych z różnymi typami danych, zmienne są automatycznie przekształcane tak, aby wyrażenie dało się w rozsądny sposób obliczyć. Język nie wymusza reguł mieszania typów wskaźników. Nie stosuje się sprawdzania podczas wykonywania programu zakresu zmienności indeksów. Nie sprawdza się także zgodności typów argumentów formalnych i argumentów aktualnych w wywołaniach funkcji, ani podczas kompilacji, ani podczas wykonywania programu.

W Unixie i w niektórych innych systemach operacyjnych istnieje program lint, służący do wykrywania wielu z tych błędów. Może to wydać się dziwne, że zadanie sprawdzenia poprawności programu zostało podzielone między dwa programy, ale odpowiada to postawionemu celowi uzyskania szybkiego i małego kompilatora.

Pliki źródłowe

Program w C jest zapisany w postaci pliku tekstowego, który później jest kompilowany do postaci pośredniej. Pliki pośrednie podlegają konsolidacji z innymi plikami tego samego typu i z bibliotekami w celu uzyskania programu wykonywalnego.

Te dwa kroki procesu (kompilacja i konsolidacja) są niezbędne nawet dla najprostszych programów, ponieważ – jak wspomniano wcześniej – bardzo wiele właściwości języka jest realizowanych przez funkcje biblioteczne, nie zaś przez wbudowane instrukcje języka C. Programista może swobodnie dodawać nowe funkcje do istniejących bibliotek albo może tworzyć nowe biblioteki.

Przechowywanie programu w postaci zbioru niewielkich plików zmniejsza czas kompilacji, gdyż pliki nie zmienione nie muszą być powtórnie kompilowane. Ponadto małe pliki łatwiej redagować.

Jeśli jakiś plik staje się zbyt duży dla programisty lub dla edytora, to można go podzielić na dwa pliki mniejsze. Zmiana w jednym z nich wymaga powtórnego kom-

pilowania tylko tego pliku; drugi nie wymaga tej operacji. Konsolidator połączy te dwa pliki z bibliotekami w celu utworzenia pliku wykonywalnego.

Pliki są włączane podczas kompilacji dyrektywą #include preprocesora.

Identyfikatory

Identyfikatory (nazwy) zmiennych i funkcji są to dowolne kombinacje liter (zarówno małych, jak wielkich) i cyfr – pod warunkiem, że rozpoczynają się od litery. W identyfikatorze można użyć także znaku podkreślenia, chociaż niektóre kompilatory nie akceptują go na początku ciągu.

Litery małe są uważane za różne od wielkich; na przykład identyfikator abc jest różny od aBc.

Przy określaniu niepowtarzalności identyfikatora nie zawsze bierze się pod uwagę wszystkie znaki; niektóre kompilatory biorą pod uwagę tylko osiem pierwszych znaków identyfikatora, inne więcej.

Oto powszechnie przyjęta konwencja tworzenia nazw w języku C:
- używa się wyłącznie małych liter, z wyjątkiem nazw stałych symbolicznych (tj. stałych określonych za pomocą dyrektywy #define preprocesora) i nazw typów utworzonych za pomocą typedef;
- używa się liter początkowych od i do n w celu oznaczenia liczb całkowitych;
- używa się litery początkowej c do oznaczenia zmiennych znakowych;
- używa się litery początkowej s do oznaczenia ciągów znaków.

Definicje danych i deklaracje

Zmienne muszą być zdefiniowane przed użyciem. Dostępne są następujące podstawowe typy danych:

Typ	Definicja	Typowy rozmiar
char	zmienna znakowa	8 bitów (bajt)
short int	liczba całkowita krótka	16 bitów
int	liczba całkowita	zależy od realizacji
long int	liczba całkowita długa	32 bity
float	liczba zmiennoprzecinkowa	32 bity
double	liczba zmiennoprzecinkowa długa	64 bity
unsigned char	zmienna znakowa bez znaku	8 bitów
unsigned int	liczba całkowita bez znaku	zależy od realizacji

Podane tu rozmiary są najczęściej używane, ale programista powinien sprawdzić rozmiar dla używanej realizacji kompilatora.

Występują jeszcze inne typy danych:
- *tablica*, czyli wielokrotne wystąpienie identycznych zmiennych;
- *struktura*, czyli grupa powiązanych zmiennych; mogą być różnych typów;

- *wskaźnik*, czyli zmienna zawierająca adres innej zmiennej;
- *unia*, czyli dwie lub więcej zmiennych zajmujących w pamięci to samo miejsce;
- *dane wyliczeniowe* (enum), czyli zmienne o ograniczonym zbiorze dopuszczalnych wartości;
- *ciąg znaków* (łańcuch), czyli tablica zmiennych znakowych typu char, zakończona znakiem NULL.

Dodatkowe typy danych, oparte na podanych powyżej, można zdefiniować korzystając ze specyfikatora typedef.

Różnica między definicją danych a deklaracją danych polega na tym, że definicja danych rezerwuje miejsce w pamięci na dane, a deklaracja nie.

Zmienne zdefiniowane lub zadeklarowane poza funkcjami są dostępne od miejsca zdefiniowania aż do końca pliku źródłowego. Zmienne zdefiniowane wewnątrz funkcji są dostępne jedynie wewnątrz tych funkcji, w których zostały zdefiniowane. Zmienne zdefiniowane wewnątrz instrukcji złożonych (bloków) są dostępne tylko wewnątrz tych instrukcji złożonych, w których zostały zdefiniowane.

Zmienne mogą należeć do jednej z czterech klas:
- static – pamięć dla nich jest rezerwowana statycznie, na początku programu;
- auto – pamięć dla nich jest rezerwowana dynamicznie, gdy funkcja jest wykonywana, i jest zwalniana, gdy kończy się wykonanie funkcji (jest to domyślna klasa pamięci dla zmiennych definiowanych wewnątrz funkcji);
- extern – pamięć dla nich jest rezerwowana w innym pliku; odwołanie rozstrzyga konsolidator;
- register – są zapamiętywane w rejestrze, o ile jest to możliwe.

Stałe

Stałe są określane w tekście programu następująco:

Typ	Składnia	Przykład
char	otoczona apostrofami	'a'
ciąg znaków	otoczona cudzysłowami	"abc"
int		
liczba ósemkowa	zaczyna się 0 (zerem)	0377
liczba szesnastkowa	zaczyna się 0x	0x3f
long int	l lub L	123L
float	z kropką dziesiętną lub w notacji naukowej	3.24 lub 3.2E–4
double	jak float	

W praktyce wszystkie zmienne typu float są uważane za zmienne typu double.

Sekwencje specjalne

Niektóre stałe są różnie definiowane w różnych realizacjach, zwykle jak następuje:

\n	nowy wiersz
\r	powrót karetki (CR i LF)
\t	tabulator poziomy
\f	nowa stronica
\v	tabulator pionowy
\b	cofnięcie o znak (bakspace)
\"	podwójny cudzysłów
\'	pojedynczy cudzysłów
\\	ukośnik (backslash)
\nnn	wartość ósemkowa

Początkowy ukośnik oznacza sekwencję specjalną. Użycie wymienionych stałych zwalnia programistę od konieczności zapoznawania się zbyt głęboko z zestawem znaków komputera.

Operatory

W języku C występuje 45 operatorów, w 15 grupach. W każdej z tych grup operatory mają ten sam priorytet. Grupy operatorów są wymienione w kolejności priorytetu, od największego do najmniejszego. W każdej grupie podano zasady łączności operatorów.

1. Operatory łączne lewostronnie

()	wywołanie funkcji	getc(stdin)
[]	odwołanie do elementu tablicy	i[17]
->	wskaźnik do elementu struktury	alt_ptr->c
	odwołanie do elementu struktury	alt.c

2. Operatory łączne prawostronnie

-	minus jednoargumnetowy	-i
++	inkrementacja	i++
--	dekrementacja	i--
!	negacja logiczna	!znaleziony
~	uzupełnienie do 1	~0x7f
*	wskazanie pośrednie	*c_ptr
&	adres elementu	&i
sizeof	rozmiar zmiennej lub typu	sizeof(j)
(*typ*)	zmiana typu	(int)c

3. Operatory łączne lewostronnie

*	mnożenie	i * j
/	dzielenie	i / j
%	operacja modulo	i % j

4. Operatory łączne lewostronnie

+	dodawanie	i + j

<table>
<tr><td>–</td><td>odejmowanie</td><td>i – j</td></tr>
<tr><td colspan="3">5. Operatory łączne lewostronnie</td></tr>
<tr><td>< <</td><td>przesunięcie bitowe w lewo</td><td>i < < 2</td></tr>
<tr><td>> ></td><td>przesunięcie bitowe w prawo</td><td>i > > 2</td></tr>
<tr><td colspan="3">6. Operatory łączne lewostronnie</td></tr>
<tr><td><</td><td>mniejszy niż</td><td>i < j</td></tr>
<tr><td>< =</td><td>mniejszy lub równy</td><td>i < = 6</td></tr>
<tr><td>></td><td>większy niż</td><td>i > 2</td></tr>
<tr><td>> =</td><td>większy lub równy</td><td>i > = j</td></tr>
<tr><td colspan="3">7. Operatory łączne lewostronnie</td></tr>
<tr><td>= =</td><td>równy</td><td>if(i = = 5)</td></tr>
<tr><td>! =</td><td>nierówny</td><td>if(i ! = 5)</td></tr>
<tr><td colspan="3">8. Operatory łączne lewostronnie</td></tr>
<tr><td>&</td><td>bitowy iloczyn logiczny</td><td>c & 033</td></tr>
<tr><td colspan="3">9. Operatory łączne lewostronnie</td></tr>
<tr><td>^</td><td>bitowa suma modulo 2</td><td>c ^ 0317</td></tr>
<tr><td colspan="3">10. Operatory łączne lewostronnie</td></tr>
<tr><td>|</td><td>bitowa suma logiczna</td><td>c | 0333</td></tr>
<tr><td colspan="3">11. Operatory łączne lewostronnie</td></tr>
<tr><td>&&</td><td>iloczyn logiczny</td><td>i = =5 && j = = 6</td></tr>
<tr><td colspan="3">12. Operatory łączne lewostronnie</td></tr>
<tr><td>| |</td><td>suma logiczna</td><td>i = =5 | | j = = 6</td></tr>
<tr><td colspan="3">13. Operatory łączne prawostronnie</td></tr>
<tr><td>? :</td><td>wyrażenie warunkowe</td><td>i >4 ? i : j</td></tr>
<tr><td colspan="3">14. Operatory łączne prawostronnie</td></tr>
<tr><td>=</td><td>przypisanie</td><td>i = 7</td></tr>
<tr><td>* =</td><td>mnożenie, potem przypisanie</td><td>i * = 3</td></tr>
<tr><td>/ =</td><td>dzielenie, potem przypisanie</td><td>i / = 4</td></tr>
<tr><td>% =</td><td>modulo, potem przypisanie</td><td>i % = 4</td></tr>
<tr><td>+ =</td><td>dodawanie, potem przypisanie</td><td>i + = 5</td></tr>
<tr><td>– =</td><td>odejmowanie, potem przypisanie</td><td>i – = 6</td></tr>
<tr><td>& =</td><td>iloczyn bitowy, potem przypisanie</td><td>i & = 0333</td></tr>
<tr><td>^ =</td><td>suma mod 2, potem przypisanie</td><td>i ^ = 0317</td></tr>
<tr><td>| =</td><td>iloczyn logiczny, potem przypisanie</td><td>i | = 0177</td></tr>
<tr><td>< < =</td><td>przesunięcie w lewo, potem przypisanie</td><td>i < < = 2</td></tr>
<tr><td>> > =</td><td>przesunięcie w prawo, potem przypisanie</td><td>i > > = 3</td></tr>
<tr><td colspan="3">15. Operatory łączne lewostronnie</td></tr>
<tr><td></td><td>oblicz i odrzuć</td><td>i = 5, j</td></tr>
</table>

Wyrażenia

Wyrażenie może przyjąć jedną z następujących postaci:
- nazwa zmiennej,
- wywołanie funkcji,
- nazwa tablicy,
- stała,
- nazwa funkcji,
- odwołanie do elementu struktury,
- odwołanie do elementu tablicy,
- jedna z powyższych postaci z nawiasami i operatorami.

Instrukcje

Wyrażenie zakończone średnikiem jest uważane za instrukcję. Na przykład:

```
i+1
```

jest wyrażeniem, podczas gdy

```
i+1;
```

jest instrukcją. Oto przykłady innych instrukcji:

1. ; instukcja pusta
2. if (*wyrażenie*) *instrukcja*
3. If (wyrażenie) *instrukcja1* else *instrukcja2*
4. while (*wyrażenie*) *instrukcja*
5. do *instrukcja* while (*wyrażenie*);
6. for (*wyrażenie1*;*wyrażenie2*;*wyrażenie3*) *instrukcja*
7. switch *wyrażenie_całkowitoliczbowe*

```
{
  case stała1: instrukcja1
  case stała2: instrukcja2
  ...
}
```

8. break;
9. continue;
10. goto *etykieta*;
11. return;
12. return *wyrażenie*;

Instrukcja złożona (lub blok) powstaje przez zamknięcie kilku instrukcji w nawiasy klamrowe { i }; na przykład

```
{
  i++;
  j++
}
```

jest instrukcją złożoną (z dwóch instrukcji prostych) i może być użyta w każdym miejscu, gdzie dopuszcza się użycie instrukcji.

Tablice i wskaźniki

Wskaźnik jest to zmienna zawierająca adres innej zmiennej. W języku C wskaźniki mają określony typ, to znaczy definiuje się wskaźnik jako wskaźnik do zmiennej typu `char`, `int` itp.

W przypadku wskaźników dopuszcza się tylko proste operacje arytmetyczne. Do wskaźnika można dodać (lub od niego odjąć) tylko liczbę całkowitą, ale nie są dopuszczalne operacje w zasadzie bezsensowne, takie jak dodawanie wskaźników, dzielenie wskaźnika przez liczbę całkowitą itp.

Dla wielu typów komputerów nie ma żadnej różnicy między wskaźnikiem a liczbą całkowitą; obydwa te obiekty zajmują tę samą przestrzeń. Jest więc możliwe operowanie nimi jak liczbami całkowitymi. Kompilator może, ale nie musi podać komunikatu o błędzie, ale wyników takich operacji nie można przenosić na inny komputer. Program zawierający takie operacje będzie pracował poprawnie na jednym komputerze, ale nie będzie pracował na innym.

Tablica jest to ciąg zmiennych tego samego typu zajmujących ciągły obszar w pamięci. Na przykład definicja

```
int i[10]
```

określa tablicę dziesięciu liczb całkowitych, o nazwach od `i[0]` do `i[9]`. Zauważmy, że w języku C pierwszy element tablicy ma numer zero.

Definicja

```
int *p_i
```

określa zmienną `p_i` jako wskaźnik do liczby całkowitej.

Wyrażenie `*p_i` oznacza "zmienna wskazywana przez `p_i`".

W języku C nazwa tablicy (bez nawiasu kwadratowego) jest rozumiana nie jako wartość pierwszego elementu tablicy, lecz raczej jako jego adres. Tak więc wyrażenie

```
p_i = i;
```

ustawia tak `p_i`, aby wskazywał pierwszy element tablicy `i`.

Jeśli `p_i` wskazuje element `i[0]`, to `p_i + 1` wskazuje `i[1]`. Tak więc inkrementacja wskaźnika nie zwiększa go o jeden bajt, ale o jeden element. Jeżeli wskaźnik został określony jako wskaźnik do elementu znakowego typu `char`, to inkrementacja oznacza "dodaj rozmiar zmiennej typu `char`". Inkrementacja wskaźnika do zmiennej typu `int` oznacza "dodaj rozmiar zmiennej typu `int`".

Innymi słowy, istnieją dwie metody odwoływania się do elementu tablicy – przez użycie indeksu w tablicy (np. `i[3]`) lub odpowiednio zdefiniowanego wskaźnika (np. `*(p_i + 3)`).

Funkcje

We wszystkich plikach źródłowych, które są kompilowane i konsolidowane w celu utworzenia programu wynikowego, tylko jedna funkcja może nosić nazwę main. Pierwsza wykonywana instrukcja w programie jest też pierwszą wykonywalną instrukcją w main.

W języku C wszystkie funkcje znajdują się na tym samym poziomie składniowym; nie można definiować funkcji wewnątrz funkcji. Co więcej, wszystkie funkcje we wszystkich plikach tworzących program oraz wszystkie funkcje w bibliotekach dołączonych przez konsolidator są widoczne dla wszystkich pozostałych funkcji. Jedynym wyjątkiem jest funkcja określona jako static; jej widoczność jest ograniczona do pliku źródłowego, w którym została zdefiniowana.

Funkcja może być wywoływana z argumentami, które są przekazywane przez wartość. W języku C nie występuje wywołanie przez nazwę, z wyjątkiem argumentów będących tablicą; w tym przypadku przekazuje się wskaźnik do tablicy, a nie samą tablicę, ponieważ – jak powiedzieliśmy wcześniej – przez nazwę tablicy rozumie się jej adres. W niektórych wersjach C jest możliwe przekazanie struktury jako argumentu; także w tym przypadku następuje przekazanie wskaźnika, czyli struktura jest wywoływana przez nazwę.

Funkcja nie może modyfikować wartości swoich argumentów (z wyjątkiem dwóch przypadków wywołania przez nazwę omówionych wyżej). Mimo to funkcja może modyfikować wartość zmiennej niewidocznej dla niej, jeśli widzi ona adres tej zmiennej jako swój argument albo jako zmienną globalną.

Zmienne definiowane wewnątrz funkcji są albo klasy auto (tworzone w chwili, gdy funkcja zaczyna być wykonywana, i znikające po zakończeniu jej wykonania), albo klasy static (tworzone w chwili, gdy program zaczyna być wykonywany, i znikające po zakończeniu wykonania), albo klasy register (zmienna klasy auto, którą kompilator stara się umieścić w rejestrze). Funkcje nie rezerwują pamięci dla zmiennych definiowanych jako extern. Żadna zmienna zdefiniowana wewnątrz funkcji nie jest widoczna poza nią.

Funkcja może podawać wynik za pomocą instrukcji return. Jeśli w nagłówku funkcji nie został podany typ podawanego wyniku, to przyjmuje się, że jest to wartość typu int. Funkcja może dać wynik innego typu niż int, ale musi ona być zdefiniowana lub zadeklarowana przed jej pierwszym użyciem w pliku źródłowym.

Możliwe jest przekazanie jako argumentu funkcji nazwy innej funkcji, zwanej "wskaźnikiem do funkcji".

Ciało funkcji, to jest definicje, deklaracje i instrukcje, musi być zawarte w nawiasach klamrowych.

Listę funkcji zwykle dostępnych w większości wersji C zamieszczono w dodatku A. Dodatek B zawiera kod źródłowy niektórych funkcji operujących ciągami znaków. Dodatek C zawiera kod źródłowy funkcji do kodowania i dekodowania plików tekstowych z użyciem hasła, a dodatek D – kod źródłowy funkcji do budowy drzew binarnych z wyrażeń wielomianowych.

Preprocesor

Pierwsza faza kompilacji programu w C następuje w preprocesorze, który usuwa komentarze, podstawia teksty i warunkowo wstawia lub usuwa części tekstu z procesu kompilacji. W systemie operacyjnym Unix (i w niektórych innych systemach) jest możliwe uruchomienie preprocesora oddzielnie; nie jest to cecha specyficzna dla języka C i występuje w innych językach.

Wiersze w tekście źródłowym poprzedzone # są dyrektywami preprocesora:

#define; realizuje parametryczne podstawianie tekstu; służy do definiowania stałych symbolicznych i makrodefinicji z parametrami;

#if *wyrażenie_stałe instrukcje* #endif; jeśli *wyrażenie* jest niezerowe, to włącza instrukcje aż do #endif do tekstu kompilowanego;

#if *wyrażenie_stałe instrukcje* #else *instrukcje* #endif; podobna do poprzedniej dyrektywy; zawiera instrukcje po #else;

#ifdef *identyfikator instrukcje* #endif;
jeśli w tej części tekstu *identyfikator* jest zdefiniowany, to włącza instrukcje aż do #endif do tekstu kompilowanego;

#ifdef *identyfikator instrukcje* #else *instrukcje* #endif; podobna do poprzedniej dyrektywy; zawiera instrukcje po #else;

#ifndef *identyfikator instrukcje* #endif; jeśli w tej części tekstu *identyfikator* nie jest zdefiniowany, to włącza instrukcje aż do #endif do tekstu kompilowanego;

#ifndef *identyfikator instrukcje* #else *instrukcje* #endif; podobna do poprzedniej dyrektywy; zawiera instrukcje po #else;

#include "*nazwa_pliku*" lub #include <*nazwa_pliku*>; włącza wskazany plik do tekstu kompilowanego;

#line *n* "*plik*"; identyfikuje następne wiersze jako pochodzące z "*pliku*", wiersz numer *n*;

#undef *identyfikator*; powoduje, że definicja *identyfikatora* nie jest już rozpoznawana.

Argumenty funkcji main

W wielu systemach operacyjnych funkcja main akceptuje dwie zmienne: argc, tj. liczbę całkowitą pokazującą liczbę argumentów w wierszu poleceń i argv, tj. wskaźnik do tablicy wskaźników znakowych, wskazujących właściwe argumenty.

Przykładowy program

Poniżej podano przykładowy program zawierający dwie funkcje, pokazujący ogólną strukturę programu w języku C.

```
#define CONSTANT1 5
#define CONSTANT2 4
```

```
main(argc,argv)
int argc;
char *argv[];
{
   int j = CONSTANT1;
   int k = CONSTANT2;
   int iloczyn;
   iloczyn = mult(j,k);
}
mult(m,n)
int m,n;
{
   return m*n;
}
```

Funkcja main tego programu nie korzysta z argumentów argc i argv. Argumenty formalne zostały zdefiniowane przed klamrą otwierającą funkcji, a zmienne lokalne funkcji są zadeklarowane po tym nawiasie otwierającym. Funkcja main ma trzy zmienne klasy auto (domyślna klasa pamięci zmiennych zawartych wewnątrz funkcji): j, k, iloczyn – wszystkie typu int.

Funkcja mult daje w wyniku wartość typu int (domyślna) i wobec tego nie jest ona deklarowana wewnątrz funkcji main. Instrukcja przypisania w main wywołuje funkcję mult z argumentami j, k i przypisuje wynik otrzymany z funkcji mult zmiennej iloczyn.

Zauważmy, że nie ma żadnego słowa kluczowego określającego początek funkcji.

Funkcja mult została zdefiniowana z dwoma argumentami formalnymi m, n, odpowiadającymi dwom argumentom aktualnym j, k, z którymi została wywołana z main. Instrukcja return oblicza iloczyn i przekazuje wynik.

3. Uruchamianie programów

W 1954 roku, Grace Murray Hopper – wtedy w stopniu kapitana Marynarki Wojennej USA – pracując na komputerze Mark II wykryła, że mól dostał się do przekaźników maszyny i spowodował przerwę w jej pracy. Usunęła ona starannie robaka, przykleiła go taśmą klejącą do dziennika pracy maszyny i powiedziała swemu szefowi, że "odrobaczyła" maszynę (ang. *debugged*, od *bug* – robak, pluskwa). Stąd pochodzi angielskie określenie na uruchamianie urządzeń i programów – *debugging*. Wspomniany dziennik, łącznie z molem, można do tej pory oglądać w Muzeum Broni Marynarki Wojennej USA w Dahlgren, w stanie Virginia.

Programowanie jest to dobrze zdefiniowana dziedzina działalności, o dobrze określonej stukturze, której można nauczyć się na uczelni lub z podręczników, a którą posługujemy się w godzinach pracy. Uruchamianie jest to bezładny, nieukierunkowany, niewdzięczny wysiłek wykonywany pod przymusem i naciskiem, często późną nocą, z poganianiem przez nie dotrzymane terminy. Uruchamianie pochłania większą część wysiłku większości programistów. Nie ma ono ani metod ani reguł i nikt nie uczy go na uczelni. Początkujący programiści muszą polegać na własnej pomysłowości i pomocy kolegów.

Pierwsze komputery, takie jak wspomniany Mark II i jego następcy z lat pięćdziesiątych, były projektowane w epoce, gdy niewiele wiedziano na temat metod tworzenia oprogramowania. Programiści byli na ogół inżynierami lub matematykami wierzącymi, że metody, jakie opanowali w swej rodzimej dziedzinie, będą im służyć również przy programowaniu. Jak się okazało, programowanie jest to dziedzina całkowicie inna, ma swoje problemy, z których wiele da się odnieść do złożoności psychiki ludzkiej.

W perspektywie wydaje się, że pierwsi inżynierowie oprogramowania nie doceniali złożoności programowania i swoich możliwości popełniania błędów. Podobnie było z twórcami sprzętu; często byli to zresztą ci sami ludzie. Przyjmowali oni, że będą musieli uruchomić komputer raz, podobnie jak raz przeprowadza się przegląd nowego samochodu. Wydaje się, że zasadnicza różnica – polegająca na tym, że samochód wykonuje te same ruchy przez cały czas, podczas gdy komputer jest zmieniającą się

ciągle maszyną, ciągle wykonującą różne programy (niektóre z nich zostały napisane w pośpiechu ledwie parę godzin temu) – umknęła uwadze tych architektów ówczesnych komputerów.

Jedna z 5-bitowych maszyn owej ery miała dokładnie 32 kody operacji; każda kombiancja bitów była ważną instrukcją komputera. Mały błąd powodował, że komputer wykonywał swoje dane jako instrukcję i gdy w końcu "padał", licznik programu wskazywał daleko od miejsca powstania problemu. Gdy coś było nie w porządku, to ówczesne komputery po prostu stawały. Nie pojawiały się komunikaty, ani jawne, ani trudne do interpretacji. Programista mógł – na podstawie stanu lampek – odczytać stan licznika rozkazów, a po przestawieniu kilku przełączników zobaczyć zawartość komórki pamięci (oczywiście dwójkowo). Dalej już był zdany na własne siły. Nie były znane asemblery, języki wysokiego poziomu ani systemy operacyjne. O symbolicznych programach uruchomieniowych nikt nie słyszał ani nawet nie marzył.

W takich warunkach od programisty oczekiwano, że będzie starannie śledził przebieg każdego wykonania swego programu, szukał błędów i jeśli je znajdzie, to poprawi program metodą, którą można określić mianem "uruchamiania na biurku", polegającą na tym, że faktyczne sprawdzanie programu odbywało się bez wykonywania go. Programista rozkładał wydruk programu na biurku (lub na podłodze) i starannie śledził jego przebieg, instrukcja po instrukcji, zapisywał zmiany wartości każdej zmiennej i rejestrów, poszukując w ten sposób błędów logicznych.

W owych czasach programista nie używał komputera jako narzędzia do testowania i uruchamiania programów – jak czyni to obecnie. W typowej instalacji z pracą w trybie wsadowym (a takie instalacje przetrwały do lat siedemdziesiątych, czasami nawet do lat osiemdziesiątych) miał on szansę skompilować program raz dziennie. Zwykle oddawał stos kart perforowanych operatorowi systemu mając nadzieję, że kompilacja zostanie wykonana w nocy. Po powrocie do pracy następnego dnia mógł przejrzeć wydruk programu, poprawić wszystkie błędy jakie mu się udało znaleźć i powtórnie oddać program do kompilacji. Programiści bardzo starannie sprawdzali swe programy na biurku, gdyż nawet najdrobniejsze błędy oznaczały stratę wielu dni pracy.

Opracowanie środków pracy konwersacyjnej radykalnie zmieniło metody programowania. Teraz tak łatwo robi się drobne poprawki w programie i wykonuje kompletny cykl kompilacji, testowania i powtórnej kompilacji, że programiści mają tendencję do testowania każdego przychodzącego im do głowy pomysłu – nawet takiego, który przy dokładniejszym przyjrzeniu się nie daje szans sukcesu. Powszechna dostępność symbolicznych programów uruchomieniowych, które pozwalają programiście śledzić wykonanie programu oraz zmieniać dane i instrukcje na poziomie kodu źródłowego, spowodowała, że sprawdzanie programu przy biurku stało się rzadkością.

Dawne warunki pracy pozwalały programiście pracować powoli i starannie. Sprawdzanie programu na biurku jest istotnie cenną i skuteczną metodą uruchamiania, ale we współczesnych warunkach wydaje się niemal anachronizmem: w czasie potrzebnym na uzyskanie kompletnego wydruku programu i rozłożenie go na biurku można przynajmniej raz wykonać pełny cykl kompilacji i albo potwierdzić, albo odrzucić hipotezę co do powodów błędnego działania programu. Niestety taki rodzaj

pracy, który można by określić mianem "najpierw strzelać, potem pytać", powoduje, że programista spędza wiele czasu na drobiazgach i nie jest w stanie wgłębić się w trudniejsze problemy, nie rozumie dobrze napisanego przez siebie programu i nie potrafi wychwycić konsekwencji błędu, którego wpływ ujawnia się daleko od miejsca jego pojawienia się.

Metoda dedukcyjna (sprawdzanie każdej hipotezy jaka przychodzi na myśl) nie jest z natury lepsza niż metoda indukcyjna (sprawdzanie na biurku). Pierwsza, mimo że pozwala szybko wskazać objawy błędu, jest zbyt wąska, aby wskazać jego przyczyny. Druga wprawdzie pokazuje przyczyny błedu, ale pozwala jedynie na zgadywanie jego skutków. Aby uruchamianie mogło być efektywne, musi być kombinacją tych obydwu metod.

Dążenie do "naukowych" metod uruchamiania

Naukowe metody badań laboratoryjnych są oparte na pomyśle, że powtarzane (i powtarzalne) eksperymenty będą dawać wyniki umożliwiające badaczowi sformułowanie testowalnych hipotez wyjaśniających te wyniki. Hipotezy można potwierdzić lub odrzucić przez przeprowadzenie bardziej złożonych eksperymentów, przenaczonych do wyizolowania wpływu poszczególnych czynników i do stwierdzenia, które z nich są istotne z punktu widzenia testowanej hipotezy, a które nie.

Ta sama metoda daje się bezpośrednio zastosować do uruchamiania. Źle działający program można wykonywać wielokrotnie ze zmienionym zestawem danych, instrukcji lub w zmienionym środowisku, w celu wykrycia, który z tych czynników ma wpływ na jakość programu. Komputer stanowi przy tym laboratorium programisty.

W procesie uruchamiania można wyróżnić cztery etapy:

- zbieranie dostępnej, ważnej informacji,
- powtórzenie doświadczenia w celu znalezienia wzorów w błędnie działającym programie,
- sformułowanie hipotezy wyjaśniającej błędne zachowanie się programu,
- próba potwierdzenia bądź obalenia hipotezy przez zmianę programu lub danych i sprawdzanie, czy problem został rozwiązany przez testowanie poprawionego programu przy użyciu dużej ilości danych rzeczywistych.

Poniżej dyskutujemy bardziej szczegółowo każdy z tych kroków.

Zbieranie dostępnej, ważnej informacji

Nie zawsze łatwe jest określenie, jaka informacja jest ważna, a często programista nie docenia ilości informacji, jaka naprawdę jest dostępna. Poleganie na sprawozdaniach innych użytkowników jest na ogół mało pewne i najlepiej dokładnie sprawdzić wszystkie te raporty. Użytkownicy–niespecjaliści mają tendencję do przesadzania z objawami błędu i często mylą objawy obecnego błędu z objawami błędu poprzedniego, a ich analiza tego, co spowodowało błąd, jest rzadko prawdziwa. Programista powinien polegać na tym co widzi własnymi oczami, pamiętając o tym, że często ludzie widzą to, co chcą widzieć, a nie to co jest w rzeczywistości.

"Jeden z urzędników zadzwonił i powiedział, że System Przetwarzania Danych błędnie sumuje oferty. Jak się okazało, urwał się jeden płatek z margerytkowej drukarki, która w efekcie nie drukowała zer."

W rzeczywistości dopiero po wykryciu błędu można z całą pewnością stwierdzić, jaka informacja była ważna, a jaka "szumem" przypadkowym. Podczas zbierania tych informacji należy wystrzegać się formułowania pospiesznych i nieudokumentowanych sądów na temat znaczenia zbieranej informacji.

Powtórzenie doświadczenia w celu znalezienia wzorów w błędnie działającym programie

Jeśli problem nie da sie powtórzyć, to nigdy nie będzie można z całą pewnością stwierdzić, że został rozwiązany – przynajmniej z punktu widzenia czysto formalnego. Z drugiej strony problem, który nigdy się nie powtarza, trudno nazwać problemem.

Większość problemów daje się wielokrotnie powtarzać i programista może za każdym powtórzeniem wyciągnąć tyle informacji, ile się da przez niewielkie zmiany danych, programu lub środowiska w celu odizolowania czynników, które zdają się mieć wpływ na zachowanie programu, od tych, które takiego wpływu nie mają. Jednak typowy program ma wiele setek instrukcji operujących dziesiątkami zmiennych w niesłychanie skomplikowanym środowisku. Jak wybrać te, które są ważne z punktu widzenia rozważanego problemu?

Pierwsze pytanie, jakie należy zadać, to "Co się zmieniło?". Jeśli program pracował dobrze w przeszłości i nagle zaczął działać źle, to coś musiało się zmienić. Czasem programista nie wie nawet, że nastąpiła zmiana, dopóki niespodziewane złe działanie programu nie wyciąga jej na światło dzienne. Ważne jest, aby programista zaakceptował stwierdzenie, że nastąpiła jakaś zmiana, niezależnie od tego, jak wiele osób usiłuje utwierdzić go w przekonaniu, że nic takiego nie wydarzyło się. Jego praca polega właśnie na znalezieniu tej zmiany.

"Pewnego dnia zadzwonił szef ośrodka obliczeniowego i powiedział, że nasz pakiet oprogramowania komunikacyjnego przestał działać. Zapytałem go, co się zmieniło, a on naturalnie odpowiedział, że nic nie uległo zmianie. Upierałem się, że coś musiało się zmienić, ale on też obstawał przy swoim. Po kilku dalszych pytaniach zdecydowałem pojechać do klienta (3 godziny jazdy). Po przyjechaniu poprosiłem o demonstrację uszkodzenia, a sam patrzyłem na modem. Ku memu zaskoczeniu żadne światełko na modemie nie zapalało się, nawet DTR (Data Terminal Ready – terminal gotów). Jak się okazało, problem polegał na tym, że modem nie był włączony do sieci."

Oczywiście może się tak zdarzyć, że program nigdy nie pracował poprawnie, ale jego złe działanie nie zostało zauważone. W tym przypadku należałoby zapytać, dlaczego tak się stało. Czy wyniki nigdy nie zostały sprawdzane wcześniej, czy też nowy

zestaw danych spowodował ujawnienie się błędu? Zauważmy, że nawet tutaj musiała wystąpić jakaś zmiana, może tylko w zwyczajach pracy użytkownika.

Upieranie się przy błędnym mniemaniu, mimo coraz bardziej przekonywających dowodów jego błędności, jest czarującą cechą ludzkiej natury, którą autorzy powieści kryminalnych wykorzystują do swoich celów, wiodąc czytelnika od jednego błędnego wniosku do drugiego, ale która doprowadziła niejednego programistę do długotrwałego błądzenia po kolejnych ślepych zaułkach.

> W swej książce *Tablica pierwiastków* Primo Levi opisuje wysiłki chemika, pracującego w fabryce wytwarzającej błony filmowe, mające na celu znalezienie dziwnego błędu w produktach firmy, ale tylko w tych, które zostały wyprodukowane w środę. Wiele pracy poświęcił on rozwiązaniu problemu, co też zmieniło się w środę, aż w końcu skojarzył to z interesującym, ale mało związanym faktem: lokalny rybak zaczął skarżyć się na nieudane połowy mniej więcej w tym samym czasie, gdy zaczęły się kłopoty fabryki. Wtedy już łatwo wykrył, że garbarnia znajdująca się kilka kilometrów w górę rzeki nad pralnią zaczęła spuszczać do rzeki zanieczyszczenia z procesu produkcyjnego. Resztki tych zanieczyszczeń dostawały się do fartuchów, dostarczanych punktualnie w każdy wtorek po południu, a w środę dostawały się do emulsji. Było ich jedynie tyle, żeby zniszczyć jednodniową produkcję i nigdy by zapewne nie znaleziono przyczyny tych strat, gdyby nie wspomniany chemik, który uparcie szukał co się zmieniło, mimo zapewnień wszystkich w pobliżu, że nic nie uległo zmianie.

Czasami problem pojawia się tylko w bardzo specyficznych warunkach. Pomocna może być próba powtórzenia doświadczenia przy prostszym lub bardziej ogólnym zestawie warunków.

Sformułowanie hipotezy wyjaśniającej błędne zachowanie się programu

Mamy nadzieję, że po dojściu do tego miejsca programista może – na podstawie objawów błędnego działania programu – określić przybliżone usytuowanie błędu w programie. Jeśli objawy zdają się wskazywać na nową część kodu, to tym lepiej, gdyż nowy program jest zawsze podejrzany, nawet jeśli objawy nie wskazują go bezpośrednio. W rzeczywistości sprawdzenie świeżo zmodyfikowanej części kodu jest wskazane niezależnie od tego, jak daleko odległy wydaje się błąd od zmiany.

Programista powinien przejrzeć podejrzany kod, poszukując następujących błędów:

- błędy grube, takie jak napisanie i++ zamiast i--;
- sposób komunikowania się funkcji między sobą;
- błąd w definicji zmiennej, powodujący, że program operuje nią w nietypowy sposób.

Doświadczenie pokazuje, że istnieje odwrotna proporcja między złożonością efektu a złożonością błędu, który go wywołał. Proste błędy, takie jak napisanie i++

zamiast ⊢ –, prowadzą na ogół do efektów bardzo trudnych do wyjaśnienia, podczas gdy naprawdę subtelne błędy dają najłatwiej identyfikowalne efekty. Jest także prawdą, że im więcej czasu spędzimy na szukaniu błędu, tym okazuje się on bardziej banalny.

Trzeba podkreślić, że "duże" błędy, błędy w projekcie systemu lub spowodowane złym planowaniem, nie poddają się wskazanym tu metodom. Ktoś powiedział, że komu nie udało się dobrze zaplanować, ten planuje zbankrutować. Nikt nie planuje poprawiania błędów, ale każdy to robi.

Przy poszukiwaniu błędów trzeba pamiętać o kilku prawidłowościach.

Ludzie mają tendencję do widzenia tego co chcą widzieć, a nie tego co jest dokładnie przed ich oczami. Jeśli sam nie widzisz błędu w kodzie, to poproś kogoś innego, aby nań popatrzył. Jest duża szansa, że ktoś nie znający programu wykryje błąd od ręki.

Jeśli od dłuższego czasu nie widzisz postępu w pracy – zrób sobie przerwę, oderwij się od problemu. Idź na spacer, na kawę lub piwo, albo jeszcze lepiej – pojedź do domu i prześpij się. Po powrocie podejdziesz do problemu ze świeżym umysłem i będziesz miał większe szanse na rozwiązanie go.

Jeśli nie możesz znaleźć tego czego szukasz tam, gdzie spodziewasz się znaleźć, to poszukaj gdzie indziej. Bez sensu jest patrzenie na ten sam kawałek kodu i mruczenie do siebie "To niemożliwe!". Jeżeli coś jest naprawdę niemożliwe, to nie powinno się zdarzyć i jeśli w kodzie, na który właśnie patrzysz, nie ma tego co by mogło spowodować to błędne działanie, to czas spojrzeć w inne miejsce kodu.

Jeśli znajdziesz jakiś błąd, nawet całkowicie nie związany z problemem, który rozwiązujesz – popraw go. Podobnie jak trudno jest znaleźć przyczynę błędnego działania programu, tak samo trudno jest przewidzieć skutki błędu w programie. Nic nie zyskuje się przez zostawienie nie poprawionego błędu w programie.

Jeśli już znalazłeś w programie błąd odpowiedzialny za problem, to nie spiesz się z powtórnym kompilowaniem. Najpierw sprawdź, czy nie ma więcej błędów.

Jeśli nie masz żadnego pomysłu co do miejsca usytuowania błędu, to zacznij od początku programu i posuwaj się w stronę końca. Możesz być pewny, że znajdziesz coś, co jest warte dokładniejszego sprawdzenia.

Jeśli nie ma nikogo, komu mógłbyś przedstawić problem (patrz wyżej), to zamknij drzwi, żeby nikt nie mógł zobaczyć, co masz zamiar zrobić, i przedstaw problem do ściany. Zacznij od początku i niczego nie pomijaj. Zmusi cię to do uporządkowania myśli i być może pozwoli odkryć coś, co do tej pory pomijałeś. Może wydać się to dziwne, ale ta metoda działa.

Próba potwierdzenia bądź obalenia hipotezy przez zmianę programu lub danych i sprawdzanie, czy problem został rozwiązany przez testowanie poprawionego programu przy użyciu dużej liczby danych rzeczywistych

Ten krok bardziej przypomina eksperymentowanie w laboratorium i wywołuje wiele podobnych kłopotów. Ważne jest, aby programista pracował systematycznie, z dobrze sprecyzowanym planem i aby był konsekwentny.

Podstawą zrozumienia działania programu jest pewien zbiór przypuszczeń, które programista przyjmuje za prawdziwe bez żadnego uzasadnienia, jedynie na wiarę. Uruchamianie wymaga od niego wyliczenia kolejno tych przypuszczeń i wskazania, które z nich są poparte faktami, a które nie. Wyniki często są zaskakujące. Jeśli programista ma do swej dyspozycji symboliczny program uruchomieniowy, to najlepsza metoda polega na prześledzeniu podejrzanych funkcji w programie instrukcja po instrukcji i poszukiwaniu tych instrukcji, które modyfikują kluczowe zmienne w nieoczekiwany sposób. Celem pracy jest zredukowanie liczby tych funkcji do jednej, a potem prześledzenie jej aż do znalezienia miejsca błędu. Jeśli taki program uruchomieniowy nie jest dostępny, to wielokrotne użycie funkcji printf w badanej funkcji jest męczącym, ale akceptowalnym jego substytutem.

Często dobrze jest zapamiętać na dyskietce wzorcową kopię programu i zbiorów danych przed wykonaniem modyfikacji.

Nigdy nie zakładaj, że rozwiązanie problemu jest na tyle poprawne, że nie wymaga dokładnego przetestowania.

> "Pewnego razu postanowiłem uzupełnić komentarze do programu. Nie zamierzałem czynić żadnych poprawek w samym programie, ale przez pomyłkę umieściłem spację między gwiazdką a kreską, co spowodowało, że kilka następnych instrukcji stało się komentarzem. Kilka tygodni później wprowadziłem drobne poprawki do programu i skompilowałem go powtórnie, w efekcie program przestał działać. Zapomniałem już o poprzednio wykonanych zmianach i spędziłem większą część dnia szukając w niewłaściwych miejscach i wpadając w wiele pułapek nim znalazłem powód".

W tym przypadku wydawało się, że nie było ani zmian, ani powodu do powstania problemu, niemniej jednak programista był zmuszony do powtórnej kompilacji i przetestowania nowej wersji.

Czasami pojawia się kilka podobnych błędów w programie i programista znalazłszy jeden przyjmuje, że znalazł jedyny jaki tam był. Jest więc zaskoczony faktem, że błąd pozostał mimo wprowadzenia poprawki. Ta sytuacja jest jeszcze jednym powodem, aby dokładnie testować wszystkie zmiany, niezależnie od tego, jak są one proste i oczywiste.

Częste błędy w C

Pisanie = zamiast = =

Wyrażenie if(i=5), mimo że poprawne składniowo w języku C, jest prawdopodobnie błędem, gdyż ma stałe wartość prawdziwą. Prawdopodobnie programista zamierzał napisać if(i= =5). Jest to błąd typowy dla programistów znających Pascal, a zaczynających programować w C.

Definiowanie tablicy n-elementowej i sięganie do elementu numer n

W języku C pierwszy element tablicy zawsze ma numer 0. Na przykład int[10] definiuje tablicę 10 liczb całkowitych od i[0] do i[9]. Chociaż element i[10] w ogóle nie występuje, to kompilator prawdopodobnie nie zasygnalizuje jako błędu próby sięgania ani do elementu i[10], ani do elementu i[1000]. Inna postać tego błędu to:

```
for(j = 0; j < = 10; j+ +)
```

Sięga się tu do elementu o jeden adres za daleko.

Niepoprawne adresowanie pośrednie

Stosując wskaźnik do wskaźnika łatwo zapomina się o napisaniu * lub pisze się o jedną * za dużo, co powoduje modyfikację niewłaściwego wskaźnika. Tak przy okazji, właściwie nigdy nie trzeba stosować więcej niż dwóch poziomów adresowania pośredniego.

Definiowanie wskaźnika bez przydzielania pamięci dla obiektu przez niego wskazywanego

Definicja char *string nie powoduje przydzielenia pamięci na ciąg znaków string, ale tylko na wskaźnik do niego. Jeżeli string jest zmienną klasy auto, to jej wartość początkowa jest nieokreślona, a wskaźnik może przypadkowo wskazywać istniejący obiekt. Inna postać tego samego błędu to:

```
char *func()
{
   char line[128];
   ...
   ...
   return line;
}
```

Funkcja func daje w wyniku wskaźnik do zmiennej znakowej, ale jej wartość jest bezsensowna, gdyż wskazuje zmienną klasy auto, która została usunięta z pamięci zanim program mógł ją pobrać.

Pisanie o jedną parę nawiasów za mało

Przykładowo, wyrażenie if(i=max(a,b)= =b) nie jest równoważne wyrażeniu if((i=max(a,b))= =b). W pierwszym przypadku i otrzymuje wartość zero lub jeden – zależnie od tego, czy max(a,b) równa się b, czy nie; w drugim przypadku i otrzymuje wartość większej z liczb a lub b. Inna postać tego samego błędu to:

```
if(fp=fopen("nazwa_pliku","r") = = NULL))
```

Poprawne jest natomiast następujące wyrażenie:

```
if((fp=fopen("nazwa_pliku","r")) = = NULL)
```

W pierwszym wyrażeniu następuje otwarcie pliku i nadanie wskaźnikowi do pliku wartości zero (co zwykle jest równoważne NULL), a w drugim – otwarcie pliku i nadanie wskaźnikowi do niego poprawnej wartości.

Czasami brakujący nawias powoduje błędną interpretację kolejności wykonywania operacji, w sposób nie zamierzony przez programistę. Na przykład, jeśli p_i jest wskaźnikiem do i, to *p_i++ powoduje zwiększenie p_i, nie i. W celu zwiększenia i należy napisać (*p_i)++.

Jako regułę należy przyjąć, że nadmiarowa para nawiasów nie szkodzi, a brakująca para nie pomaga.

Błąd w wierszu wychodzącym poza ekran

Złą praktyką programową jest "ukrywanie" części wiersza poza ekranem, gdyż jeśli w tej części zdarzy się błąd, to szanse, że się go wykryje, są małe.

Błędnie zakończone komentarze

Zakończenie wiersza z komentarzem znakami * / (to jest ze spacją między * a /) powoduje, że następujący po nim tekst, aż do poprawnie napisanego końca komentarza (to jest do */), zostaje zamieniony na komentarz i nie zostaje to wykryte przez kompilator. Jest to błąd szczególnie trudny do znalezienia.

Zmiana w funkcji bibliotecznej

Każdy programista może wprowadzić zmiany do funkcji bez wiedzy innych programistów. Taki rodzaj błędu jest na ogół szybko wykrywany, ponieważ ludzie chętnie obwiniają innych za swoje kłopoty.

Użycie złej wersji pliku

Ten błąd może przyjmować wiele postaci:

- poprawiona wersja pliku źródłowego nie zostaje zapamiętana; potem wykonuje się kompilację i próbuje uruchomić program posługując się dawną wersją pliku i otrzymuje się te same błędy;
- nie zauważamy, że poprawiona wersja nie dała się skompilować i próbujemy ponownie uruchomić dawną wersję;
- kompilujemy nową wersję, ale konsolidujemy dawną;
- uruchamiamy program z dawnym zestawem danych zamiast z nowym;
- wydruk programu, na którym pracujemy, nie odpowiada aktualnemu programowi źródłowemu;
- popełniamy różne kombinacje powyższych błędów.

Przyjmowanie, że plik danych zawiera dokładnie to, czego się spodziewasz

Przy uruchamianiu programów wszystkie założenia a priori są podejrzane; ważne są tylko fakty.

Niezgodności między argumentami formalnymi a aktualnymi

Kompilator języka C nie sprawdza, czy argumenty formalne i aktualne odpowiadają sobie zarówno co do typu, jak co do liczby, przez co często można popełnić błąd tego rodzaju. W systemie Unix można użyć programu lint do zlokalizowania takich błędów, ale to też nie jest idealne rozwiązanie, gdyż program ten wskazuje zbyt dużo błędów; trzeba też pamiętać, aby go w ogóle użyć (program lint nie jest częścią kompilatora).

Opuszczenie nazwy funkcji w wywołaniu

Szczególnie frustrującym rodzajem błędów, wynikających z "przyzwalającej" natury języka C, jest opuszczenie nazwy funkcji w jej wywołaniu bez spowodowania pojawienia się błędu składniowego, mimo że znaczenie wyrażenia staje się całkowicie inne. Na przykład wyrażenia printf("i = %x",i) i ("i = %x",i) są poprawnymi wyrażeniami języka C (w tym ostatnim przypadku jest to użycie operatora przecinkowego, a samo wyrażenie nic nie robi).

Zmiana wartości stałej nie we wszystkich miejscach

Przyjmijmy na przykład, że tablica ma 10 elementów i że w programie występują trzy pętle, w których odwołujemy się do elementów tej tablicy. Jeśli programista decyduje się zwiększyć rozmiar tablicy do 20 elementów i nie uczyni tego we wszystkich trzech miejscach, to mogą zdarzyć się dziwne rzeczy.

Najlepsza metoda uniknięcia takich kłopotów polega na zdefiniowaniu stałej symbolicznej i używaniu jej konsekwentnie zamiast tej stałej, którą ona reprezentuje, w sposób podany poniżej:

```
#define ARRAY_SIZE 10
int i[ARRAY_SIZE];
for(j = 0; j < ARRAY_SIZE; j++)
```

Gdy zwiększymy rozmiar tablicy do 20 elementów, to trzeba jedynie zmienić dyrektywę #define do postaci:

```
#define ARRAY_SIZE 20
```

Zapomnienie przecinka w wywołaniu funkcji, co powoduje zamianę operatora jednoargumentowego na operator dwuargumentowy

Trzy operatory (*, &, –) mają różne znaczenie, zależnie od tego czy są operatorami dwuargumentowymi (mnożenie, iloczyn logiczny bitowy, odejmowanie), czy jednoargumentowymi (wskazanie, adres obiektu, negacja). Jeśli i, j są zmiennymi typu int, to wywołanie funkcji func(i,&j) ma całkowicie różne znaczenie od wywołania funkcji func(i &j). W pierwszym przypadku wywołanie ma dwa argumenty: i oraz adres j, w drugim w wywołaniu podaje się tylko jeden argument, będący bitowym iloczynem logicznym i oraz j.

4. Styl programowania

Przy pisaniu programów w C zwykle dąży się do możliwie efektywnego wykorzystania następujących zasobów:
- pamięci;
- czasu wykonania programu;
- kosztów opracowania programu;
- kosztów konserwacji programu.

Jak zawsze w sytuacjach złożonych muszą istnieć pewne kompromisy między tymi czynnikami. Na przykład, napisanie programu w języku asemblerowym umożliwia efektywne wykorzystanie pamięci i uzyskanie krótkiego czasu wykonywania, ale zwiększone koszty opracowania i konserwacji takiego programu z pewnością pochłoną uzyskane oszczędności.

W tym rozdziale dyskutujemy metody osiągnięcia efekytwnego wykorzystania tych dwóch ostatnich zasobów, to jest uzyskania możliwie małych kosztów zarówno konserwacji, jak i opracowania programu, innymi słowy – optymalnego wykorzystania czasu pracy programisty przez stosowanie się do kilku – wynikających ze zdrowego rozsądku – reguł.

Dąż do prostoty

Warto jest poświęcić trochę trudu, aby wykonać pracę w możliwie najprostszy sposób. Skomplikowane procedury są trudne do zrozumienia, trudne do zmiany i dobrego udokumentowania, a co najważniejsze trudne do użycia.

Nie daj się złapać w pułapkę "robienia nieporządnie, byle szybko" w nadziei, że kiedyś w przyszłości to się uporządkuje. Takie podejście można sprowadzić do stwierdzenia: "nigdy nie ma dość czasu, aby coś zrobić porządnie, ale zawsze jest dość czasu, aby zrobić to od nowa".

Program w trakcie opracowywania jest podstawą do zmian

Przy pisaniu programu trzeba zawsze pamiętać, że ktoś – być może ty sam – będzie musiał w bliższej lub dalszej przyszłości wprowadzać do niego małe lub duże zmiany.

Staraj się pisać programy modularne, unikaj subtelnych i złożonych zależności i pisz jasno, aby inni byli w stanie cię zrozumieć.

Nie używaj zmiennych globalnych

Idealna liczba zmiennych globalnych to zero, ale w praktyce trudno osiągnąć ten cel. W każdym razie nie używaj zmiennej globalnej, jeśli wystarczy lokalna. Zadaj sobie trud podania dodatkowego argumentu, nawet jeśli oznacza to użycie wskaźnika.

Używaj nawiasów w złożonych wyrażeniach

Nawet jeśli jesteś pewny, że reguły następstwa operatorów gwarantują obliczenie wyrażenia w zamierzony sposób, to wstaw nawiasy. Osoba, która będzie musiała konserwować twój program może nie być tak mądra jak ty. Ponadto, drobna zmiana może kompletnie zmienić znaczenie źle sformułowanego wyrażenia.

Używaj dużo komentarzy

Komentarze ma się za darmo. Nie sądź, że twój kod jest oczywisty i samo-dokumentujący lub że nie będziesz miał kłopotów ze zrozumieniem go za rok.

Nie stosuj skomplikowanych warunków

Podziel zbyt skomplikowane wyrażenie if na kilka prostszych. Skomplikowane warunki są trudne do zrozumienia i modyfikowania.

Korzystaj z funkcji bibliotecznych, jeśli są dostępne

Jeśli funkcja biblioteczna nie robi dokładnie tego, czego potrzebujesz, to spróbuj mimo wszystko znaleźć sposób jej użycia, nawet jeśli będzie to wymagać napisania dodatkowych instrukcji. Funkcje biblioteczne są na ogół niezawodne, dobrze udokumentowane i nie zmieniają się, co jest ich poważną zaletą.

Używaj dobrze wybranych nazw zmiennych

Używanie długich i sensownych nazw zmiennych oznacza, że trzeba więcej pisać, ale program jest łatwiejszy do zrozumienia.

Zostaw oddzielny wiersz dla nawiasów klamrowych

Pisz program zawsze tak, aby nawias otwierający i zamykający były umieszczone w tej samej kolumnie, co pozwala łatwo sprawdzić, czy któregoś z nich nie brakuje lub czy pojawił się nadmiarowy. Nie stosuj metody spopularyzowanej w niektórych publikacjach, w której nawias otwierający jest umieszczany na końcu wiersza. Taka metoda stwarza iluzję oszczędzania miejsca, ale programista, który ją stosuje, sam szuka kłopotów.

Używaj stałych symbolicznych

Stałe, które mają nawet bardzo niewielkie szanse zmienić się, powinny być kodowane jako stałe symboliczne. Z drugiej strony nie specjalnego sensu poniższa definicja:

```
#define JEDEN 1
```

Staraj się, aby funkcje były małe

Jeżeli funkcja nie mieści się na stronie druku, to jest prawdopodobnie za duża i powinna być podzielona.

Stosuj wcięcia do uwypuklania znaczenia

Stosowanie konsekwentnej metody wcinania tekstu programu nadaje mu strukturę łatwą do zrozumienia. Oczywiście żaden kompilator nie zwraca na wcięcia uwagi.

Wyróżniaj koniec funkcji

W języku C nie występuje żadne słowo kluczowe do wyróżnienia początku lub końca funkcji; korzystaj z wiersza komentarzy lub gwiazdek w celu ich wyróżnienia, co pozwoli ci łatwo je odnaleźć. Przyjrzyj się programom w dodatkach.

Nie używaj instrukcji goto

Najlepiej jest nawet nie myśleć o użyciu instrukcji goto, ale czasami nie da się (tej myśli) uniknąć.

Naucz się idiomów i używaj ich

Popatrz na przykłady idiomów w części leksykalnej.

5. Organizacja projektu programowego w języku C

Gdy projekt programowy staje się zbyt duży dla jednej osoby i dołącza się druga osoba, to czasem stwierdza się z rozczarowaniem, że osiągnięto tylko nieznaczny wzrost efektywności. Dwie osoby wcale nie pracują dwa razy szybciej niż jedna. Zjawisko to jest powszechnie znane.

Większą część straconego czasu pochłania komunikacja. Problem, który pojedynczy programista może rozstrzygnąć samodzielnie, teraz będzie wymagać wyjaśnień (prawdopodobnie na piśmie) przeznaczonych dla innych oraz oceny ich opinii i sugestii. Jedna osoba musi czekać aż inne zakończą ważną część nim będzie mogła posunąć się do przodu z własną pracą. Trzeba będzie więcej dokumentacji. Trzeba będzie pisać sprawozdania i organizować spotkania. Trzeba będzie dopasować różne style programowania. Wszystko to powoduje, że efektywność pracy grupowej nie jest równa sumie efektywności pracy tworzących ją osób.

W tym rozdziale poruszamy problem takiej organizacji pracy w grupie, aby najlepiej wykorzystać czas programisty. Poniższe uwagi, oparte na doświadczeniu, mogą okazać się użyteczne.

Poważnie rozważ perspektywę zakupu kodu źródłowego funkcji niskiego poziomu

Wiele firm dostarcza kod źródłowy funkcji operujących plikami i funkcji ekranowych, dostatecznie uniwersalnych aby były użyteczne w wielu różnych projektach. Istotną przewagą zakupu nad pisaniem własnych funkcji jest oszczędność czasu, ale trzeba mieć pewność, że są to rzeczywiście te funkcje, które są nam potrzebne, że działają tak jak je opisano i że sprzedawca będzie w stanie dostarczać nam uzupełnienia i poprawki.

Napisz najpierw funkcje najniższego poziomu i umieść je w bibliotece

Trzeba w jakiś sposób obalić to przekonanie programistów, że nikt nie pisze tak dobrze programów jak oni i dlatego każdy z nich robi wszystko sam. Należy ich

zachęcać do korzystania z bibliotek, jeśli tylko są dostępne, zamiast pisania samemu nieznacznie tylko różniących się wersji tych samych funkcji.

Opracuj standardy na wszystko

Pozostawieni samym sobie programiści są w stanie wymyśleć niesłychanie dużo sposobów robienia tych samych rzeczy i wszystko to w celu rozwijania inwencji twórczej. Jedną z pierwszych rzeczy, jakie powinien ustalić kierownik projektu, to standaryzacja nazw zmiennych, funkcji, bibliotek i stałych symbolicznych, zwłaszcza tych, które podają ograniczenia (na przykład największy rozmiar parametru).

Należy także ustalić właściwą strukturę katalogów. Oznacza to, że każdy programista powinien pracować we własnym katalogu, z ograniczonym dostępem do innych katalogów. Powszechne stosowanie plików nagłówkowych (z rozszerzeniem .h) umożliwi pracę z tymi samymi stałymi symbolicznymi.

Unikaj sytuacji, gdy dwie osoby pracują nad tym samym programem

Programiści są ludźmi twórczymi, zostawiającymi ślad swej indywidualności na wszystkim co robią. Jeśli dwóch programistów pracuje nad tym samym programem, to niewątpliwie pojawią się swary, niezgodności i bałagan, co z pewnością przeważy nad zyskami takiej współpracy.

Ustal odpowiedzialnego za wprowadzanie reguł

Doświadczenie pokazuje, że standardy programowania częściej są łamane niż zachowywane. Powszechne tłumaczenie jest takie, że jest to sytuacja jedynie chwilowa i że zostaną one wprowadzone, gdy tylko czas pozwoli. Zamiast tego – jeśli tylko do takiej sytuacji dopuścimy – te odmienności stają się nieudokumentowaną i trudną do zrozumienia normą.

Kierownik projektu powinien wybrać szanowanego, cierpliwego i obdarzonego autorytetem członka zespołu, który będzie wymuszał stosowanie uzgodnionych standardów

6. Leksykon

Ten rozdział zawiera hasła w porządku alfabetycznym, z wyłączeniem dyrektyw preprocesora (zaczynających się od #), które umieszczono w następnym rozdziale.

Przed użyciem leksykonu najpierw przejrzyj skorowidz w celu znalezienia hasła, które najlepiej opisuje przedmiot twoich zainteresowań, a potem przejrzyj hasło w leksykonie.

Niektóre z haseł leksykonu zawierają odnośniki do innych haseł, zawierających dodatkowe informacje na zbliżony temat.

---

adres zmiennej

Do pobrania adresu zmiennej używa się jednoargumentowego operatora & (nie mylić z dwuargumentowym operatorem & – bitowym iloczynem logicznym). Jednoargumentowy operator & nie może być stosowany do zmiennych rejestrowych, pól bitowych ani tablic, gdyż nazwa tablicy nie jest zmienną (l-wartością), ale adresem, to jest stałą.

Jednoargumentowy operator * (nie mylić z dwuargumentowym operatorem *, który oznacza mnożenie) jest używany do wskazywania, to znaczy że jego argument (który musi być zmienną wskaźnikową) jest adresem zmiennej.

PRZYKŁAD

```
main()
{
    register i;
    char c[10];      /* definiuje tablicę 10-znakową */
    int j;
    int k;
    int *p_k = &k;   /* definiuje wskaźnik do liczby całkowitej i nadaje mu
                        wartość początkową równą adresowi k */
    func(&j,&p_k)    /* wywołanie funkcji z dwoma argumentami:
                        1) adresem j ;
                        2) adresem p_k (to jest adresem adresu k) */
}
    func(q,r)
    int *q;       /* wskaźnik do liczby całkowitej */
```

```
int **r;      /* wskaźnik do wskaźnika do liczby całkowitej */
{
    *q = 0;
    **r = 0;
}
```

W programie &j oznacza adres j.

Funkcja func jest wywołana z dwoma argumentami j i p_k. Proszę zwrócić uwagę na deklarację zmiennej q jako wskaźnika do liczby całkowitej i zmiennej r jako wskaźnika do wskaźnika do liczby całkowitej. Instrukcja

```
*q = 0;
```

nadaje wartość zero zmiennej całkowitoliczbowej (j w funkcji main), której adres wskazuje zmienna q. Zmienna r wskazuje zmienną p_k, która z kolei wskazuje k.

Instrukcja

```
**r = 0;
```

nadaje wartość zero zmiennej całkowitoliczbowej (k w funkcji main), której adres (podany przez p_k w funkcji main) wskazuje zmienna q.

Wyrażenia &j oraz &c[0] są poprawne, podczas gdy &c oraz &i są błędne. Zauważmy, że &c[0] i c są równoważne, ale trzeba odróżnić c[0], która jest zmienną i l-wartością, od c – stałej.

---

agregat

Agregat jest to obiekt złożony z innych obiektów, zwanych jego elementami. Przykładami agregatów są: struktura, unia i tablica. Nie jest możliwe zadanie wartości początkowej unii ani agregatowi klasy auto.

PATRZ TAKŻE

struktura
unia
tablica
inicjowanie

---

anachronizmy

Anachronizm (w znaczeniu jakie nadaje mu się w języku C) jest to intrukcja, której składnia została zmieniona od czasów najwcześniejszych wersji C i którą kompilator sygnalizuje jako błędną. Spotyka się je czasami w dawniejszych programach. Są to instukcje dwóch rodzajów.

1. Składnia niektórych operatorów typu *operator*= (na przykład += lub |=) została odwrócona z postaci =*operator* (na przykład =+ lub =|). Wcześniejsza wersja była bardziej intuicyjna, ale niestety nie była jednoznaczna, ponieważ instrukcja

```
i=-1;
```

mogła być równie dobrze rozumiana jako "przypisz –1 zmiennej i" lub "przypisz i–1 zmiennej i". Taka niejednoznaczność była szczególnie niebezpieczna w podstawieniach makro.

2. We wcześniejszych wersjach C, operator = nie był konieczny do inicjowania; na przykład można było napisać int i 0, podczas gdy współczesna wersja języka wymaga int i=0;. Znak równości stał się obowiązkowy z tego powodu, że inicjowanie postaci:

```
int i (-4);
```

jest tak podobne do definicji funkcji, że kompilatory mają kłopoty z ich rozróżnieniem (kompilator musi odczytać całe wyrażenie nim będzie w stanie je odróżnić). Także tutaj istnieje duża szansa błędów w podstawieniach makro.

Program lint, dostępny w systemie operacyjnym Unix, wskazuje takie wyrażenia jako błąd.

---

argc i argv

Większość realizacji kompilatorów języka C dopuszcza użycie dwóch argumentów funkcji main:
- int argc, zawierającego liczbę argumentów w wierszu poleceń przy wywoływaniu programu (łącznie z nazwą programu),
- char *argv[], będącego wskaźnikiem do tablicy ciągów znaków, zawierających arumenty z wiersza poleceń; alternatywnie argv może być zadeklarowany jako char **argv, jeśli programiście ta postać bardziej odpowiada.

PRZYKŁAD

Jeśli program został wywołany z systemu operacyjnego za pomocą następującego wiersza poleceń

```
proga abc def g h i
```

to argc wynosi 6, a argv jest tablicą 6 ciągów znaków:

argv[0] wynosi "proga"
argv[1] wynosi "abc"
argv[2] wynosi "def"
argv[3] wynosi "g"
argv[4] wynosi "h"
argv[5] wynosi "i"

Standardowa postać programu do przeglądania argumentów funkcji main (oprócz pierwszego, który jest zawsze nazwą programu) ma postać:

```
main(argc,argv)
int argc;
char **argv;
{
   while (--argc >0)
```

```
        {
        ...  *(++argv) ...
        }
}
```

---

argument formalny i aktualny

Argumentami formalnymi funkcji są te argumenty, które zostały zdeklarowane w jej definicji. Argumenty przekazywane funkcji podczas jej wywołania są jej argumentami aktualnymi i muszą odpowiadać co do liczby i typu argumentom formalnym – mimo, że kompilator języka nie sprawdza tej zgodności (tego rodzaju błędy wykrywa program lint).

PRZYKŁAD

```
int q;          /* definicja zmiennej */
func(i,c,s)     /* definicja funkcji */
int i;
char c;
char *s;
```

W wywołaniu funkcji:

```
func(q, 'x',"abcdefg");
```

i, c oraz s są argumentami formalnymi; q, 'x' oraz "abcdefg" są argumentami aktualnymi.

Jeśli argument aktualny nie jest tego samego typu co argument formalny, to argumentowi aktualnemu nie może być nadana poprawna wartość początkowa. Na przykład, jeśli argument aktualny jest typu double, a formalny typu short int, to na stos będzie wysłanych więcej danych niż zostanie z niego odczytanych przez funkcję.

PRZYKŁAD

Przy poniższych definicjach:

```
double k;       /* definicja zmiennej */
int i;
func(i,j)       /* definicja funkcji */
int i,j;
```

wywołanie funkcji

```
func(k,l);
```

w typowym komputerze spowoduje umieszczenie na stosie 64 bitów dla liczby k, ale funkcja pobierze z tego tylko 16 bitów dla i. Wartości i oraz j (która następuje za i na stosie) będą teraz nieprawdziwe.

PATRZ TAKŻE

wywołanie przez nazwę i przez wartość

---

arytmetyka wskaźników

Do wskaźnika można dodać (lub od niego odjąć) wyrażenie, które w wyniku daje liczbę całkowitą. Można także odjąć dwa wskaźniki. Wszystkie inne operacje na wskaźnikach (takie jak mnożenie wskaźnika przez liczbę całkowitą lub dzielenie dwóch wskaźników przez siebie) są bez sensu i nie są dopuszczalne.

PRZYKŁAD

Jeśli p, q są wskaźnikami, to dopuszczalne są następujące operacje:

```
p++
q-4
q + sizeof(int)
p-q
```

Poniższe operacje są niedopuszczalne:

```
p/2
p*q
```

---

białe znaki

Termin białe znaki (lub białe spacje) oznacza te znaki, które w druku pojawiają się jako przestrzeń nie zadrukowana – stąd ich nazwa. Należą do nich: spacja, tabulatory poziomy i pionowy, powrót karetki i znak nowego wiersza. Funkcja isspace daje wynik TRUE dla takich znaków jako jej argumentów.

Kompilator języka C pomija białe znaki wszędzie, oprócz ciągów znaków w cudzysłowach i pojedynczych znaków w cudzysłowach.

---

biblioteka

Biblioteka jest to zbiór często używanych, skompilowanych funkcji. Biblioteki zapewniają każdemu programiście prosty sposób używania funkcji opracowanych przez innych i pozwalają uniknąć "wymyślania koła" na nowo.

W przeciwieństwie do niektórych innych języków programowania, użycie bibliotek jest szczególną cechą języka C, mimo że biblioteki nie są częścią definicji języka. Twórca każdej wersji kompilatora musi dostarczyć bibliotekę "standardową", zawierającą wszystkie "standardowe" funkcje (mimo że nie ma ścisłej definicji, jakie powinny być te standardowe funkcje; większość implementacji dostarcza biblioteki w mniejszym lub większym stopniu równoważne bibliotekom dostarczanym przez jakąś ze "standardowych" wersji Unixa).

Oprócz biblioteki lub bibliotek (czasami funkcje matematyczne są dostarczane w oddzielnej bibliotece) dostarczonych przez twórcę kompilatora, programista może łatwo tworzyć własne biblioteki lub dodawać funkcje do biblioteki standardowej. Sposób wykonywania tej operacji zależy od implementacji i systemu operacyjnego. Gdy już funkcja znajduje się w bibliotece, program może z niej korzystać tak jakby była umieszczona w samym programie. Wydaje się, że swą dużą popularność system Unix i język C zawdzięczają łatwości użycia programów i bibliotek.

---

blok patrz *instrukcja złozona*

---

błąd przesunięcia o jeden

Przy przetwarzaniu tablic wiele błędów powstajc na granicy tablicy, to jest przy pierwszym lub ostatnim jej elemencie. Najbardziej popularny jest błąd przesunięcia o jeden postaci

```
int i[10];
for (i = 0; i < =10; i+ +),
```

polegający na tym, że przetwarza się o jeden element więcej niż zawiera tablica. Tablica i ma 10 elementów o numerach od 0 do 9. Nie ma elementu i[10]. Poprawna postać pętli jest następująca:

```
for (i = 0; i <10; i+ +)
```

Inna postać błędu przesunięcia o jeden to nieprzewidzenie pustego ciągu znaków, to znaczy domyślne przyjęcie, że ciąg znaków zawiera co najmniej jeden element.

---

break

Instrukcja break jest używana do natychmiastowego opuszczenia pętli for, do .. while lub instrukcji switch.

PRZYKŁAD

```
while(TRUE)
    {
       k+ +;
       if (k = = q)
         break;
       q--;
    }
j--;
```

W razie wykonania instrukcji break, następną wykonywaną instrukcją będzie j--;. Dla porównania pokazujemy poniżej użycie instrukcji continue.

```
while(TRUE)
    {
       k+ +;
       if (k = = q)
         continue;
       q--;
    }
j--;
```

W razie wykonania instrukcji continue, następną wykonywaną instrukcją będzie k+ +;.

PATRZ TAKŻE
continue

---

case patrz *switch*

---

ciąg znaków

Ciąg znaków jest specjalnym przypadkiem tablicy, a konkretnie tablicy znaków. Ciąg znaków zawsze kończy się znakiem '\0'. Funkcja strlen daje w wyniku liczbę znaków w ciągu, bez końcowego znaku '\0'.

Istnieją dwie metody definiowania ciągu znaków.

1. Definicja char str[3] rezerwuje miejsce w pamięci na 3 znaki, co wystarcza na umieszczenie tam ciągu złożonego z dwóch znaków oraz znaku kończącego '\0'. Stała str jest wskaźnikiem do pierwszego elementu tablicy i może być używana jako argument funkcji operujących ciągami znaków, wymienionych w dodatku A. Znaki w ciągu można wskazywać na przykład tak: str[2] albo *(str + 2). Zauważmy, że &str jest wyrażeniem niepoprawnym, ale &str[0] jest wyrażeniem poprawnym.

2. Definicja char *str rezerwuje miejsce w pamięci na wskaźnik, ale nie rezerwuje miejsca na ciąg znaków. Miejsce to można otrzymać przez wywołanie jednej z funkcji przydziału pamięci, na przykład malloc. Tutaj str jest zmienną, to jest l-wartością. Znaki w ciągu można wskazywać na przykład tak: *(str + 2). Wyrażenie &str jest poprawne.

Nadanie wartości początkowych jest różne w obydwu przypadkach.

1. char str[4] = {'a', 'b', 'c', 'd'};

Tutaj str jest tablicą znaków i nadaje się jej wartość początkową zgodnie z regułami nadawania wartości początkowej tablicom. Zwróćmy uwagę apostrofy wokół każdej stałej literowej.

2. char *str = "abc";

Kończące '\0' zostanie dodane przez kompilator. Zwróćmy uwagę na cudzysłowy wokół ciągu znaków.

Zauważmy, że w tym ostatnim przypadku definiuje się dwa obiekty: wskaźnik (str) i wskazywany przez niego ciąg znaków, któremu nadaje się wartość początkową.

Ciąg znaków można kontynuować w następnym wierszu w ten sposób, że należy wiersz zakończyć ukośnikiem \, za którym natychmiast następuje znak nowego wiersza. Na przykład

```
char *str = "abcdeghijklmno\
             pqrst";
```

Zauważmy, że wszystkie ciągi znaków, nawet jeśli nadano im tę samą wartość początkową, są różne.

Przy operowaniu ciągami znaków można korzystać z funkcji standardowych, opisanych w dodatku A. Zwróćmy uwagę, że wyrażenie

```
string1 = string2;
```

przypisuje wartość jednego wskaźnika drugiemu, podczas gdy

```
strcpy(string1,string2);
```

kopiuje zawartość jednego ciągu znaków do drugiego.

Instrukcja w rodzaju

```
string = "";
```

jest prawdopodobnie błędem, gdyż modyfikuje wartość zmiennej string tak, aby wskazywała adres, pod którym kompilator przechowuje znak '\0', bez żadnej informacji co znajduje się dalej za tym znakiem. Jeśli następna instrukcja będzie miała postać:

```
strcpy(string, "abc");
```

to są duże szanse że coś zostanie zniszczone.

Poniższy fragment programu jest także błędny.

```
char *func()
{
    char *string = "abcdef";
    return string;
}
```

Ponieważ zmienna string jest klasy auto, więc przestaje istnieć, gdy funkcja kończy się wykonywać, i wartość jaką ta funkcja daje w wyniku nie może wskazywać nic użytecznego. Alternatywa polega na uczynieniu zmiennej string klasy static albo na użyciu jednej z funkcji przydziału pamięci, na przykład malloc w celu nadania jej wartości. W tym ostatnim przypadku, nawet jeśli zmienna string już nie istnieje, to obszar pamięci przez nią wskazywany zachowuje swą wartość, która jest wskazywana przez wartość zwracaną przez funkcję func.

---

continue

Instrukcja continue służy do wcześniejszego zakończenia kolejnej iteracji pętli for, do .. while lub while. Wykonanie pętli jest nadal kontynuowane.

PRZYKŁAD

```
while (TRUE)
    {
        k++;
        if (k == q)
            continue;
        q--;
    }
j--;
```

W razie wykonania instrukcji continue, następną wykonywaną instrukcją będzie k++;. Dla porównania pokazujemy poniżej użycie instrukcji break.

```
while(TRUE)
   {
      k++;
      if (k == q)
         break;
      q--;
   }
j--;
```

W razie wykonania instrukcji break, następną wykonywaną instrukcją będzie j--;.

PATRZ TAKŻE
break

---

czas życia zmiennej
Czas życia zmiennej jest to ta część działania programu, w której zmienna istnieje. Przy takim rozumieniu tego pojęcia, zmienne dzielą się na dwie grupy.

1. Zmienne istniejące przez cały czas pracy programu. Wszystkie zmienne zdefiniowane poza funkcjami zajmują obszar pamięci przydzielony przez kompilator, podobnie jak zmienne klasy static, zdefiniowane wewnątrz funkcji.

2. Zmienne istniejące tylko w czasie wykonywania funkcji, w których zostały zdefiniowane. Wszystkie zmienne klasy auto zajmują obszar pamięci przydzielany dynamicznie za każdym razem, gdy funkcja, w której są zdefiniowane, zaczyna być wykonywana i zwalniają ten obszar po zakończeniu działania funkcji. Z tego powodu uważa się, że przed zainicjowaniem zawierają one wartości przypadkowe ("śmieci"). Programista nie może przyjmować, że zmienna klasy auto "pamięta" wartość od ostatniego wywołania funkcji.

Zauważmy, że czas życia i zasięg zmiennej są to całkowicie odmienne pojęcia. Zasięg zmiennej jest wyznaczany w czasie kompilowania programu, a czas jej życia jest związany z jego wykonaniem. Na przykład, zasięg zmiennej klasy static zdefiniowanej wewnątrz funkcji jest ograniczony do tej funkcji; jej czas życia jest równy czasowi wykonania programu.

PATRZ TAKŻE
zasięg

---

definicja funkcji patrz *funkcja*

---

definicja zmiennej
Składnia definicji zmiennej (która rezerwuje dla niej miejsce w pamięci) jest następująca:

klasa_pamięci typ nazwa_zmiennej

Lista dopuszczalnych klas pamięci jest podana w haśle klasa pamięci. Listę dopuszczalnych typów podano w haśle typy danych. Wyjaśnienie dopuszczalnych nazw zmiennych podano w haśle identyfikatory.

Za słowem *nazwa_zmiennej* może następować inicjowanie. Szczegóły dotyczące tej operacji podano w haśle inicjowanie.

PRZYKŁAD
Definicje

```
int j = 0;
char c;
char *info;
```

określają odpowiednio j jako liczbę całkowitą o wartości początkowej równej zeru, c jako znak i info jako wskaźnik do ciągu znaków.

Definicja	Definiuje
int i	i jako liczbę całkowitą
int *p	p jako wskaźnik do liczby całkowitej
int *p[4]	p jako tablicę 4 wskaźników do liczb całkowitych
int (*)p[3]	p jako wskaźnik do tablicy 3 liczb całkowitych

---

deklaracja extern
Deklaracja extern oznacza, że wymieniona zmienna została zdefiniowana gdzie indziej, zwykle w innym pliku źródłowym. W tym przypadku kompilator nie może sam wstawić adresu odwołania, gdyż tylko jeden plik źródłowy jest kompilowany na raz – operację tę musi wykonać konsolidator.

Deklaracja extern nie może zawierać inicjowania. Deklaracja extern podlega tym samym regułom co inne deklaracje.

Zmienna może być zdefiniowana globalnie tylko w jednym pliku źródłowym (w przeciwieństwie do deklaracji extern), którego postać pośrednia musi być dołączona w procesie konsolidacji. Wszystkie funkcje w programie odwołujące się do tej zmiennej klasy extern, odwołują się dokładnie do tego samego obiektu.

PRZYKŁAD
plik źródłowy nr1

```
int i;
chr c;
func_a()
{


}
```

plik źródłowy nr 2

```
extern int i;
func_b()
{
    extern char c;
    ..
    ..
}
```

Definicja zmiennej i w *pliku źródłowym nr 1* jest definicją klasy extern, to znaczy, że jej nazwa jest znana konsolidatorowi, który może rozstrzygnąć, że jest to ten sam obiekt, wskazywany przez deklarację extern w *pliku źródłowym nr 2*.

Gdyby i było zdefiniowane (jako int i) w *pliku źródłowym 2* zamiast być tam deklarowane (jako extern int i), to występowałyby w programie dwie zmienne o nazwie i, każda o innym zasięgu.

PATRZ TAKŻE

zasięg

---

deklaracje funkcji

Jeśli funkcja w wyniku daje wartość inną niż liczba całkowita int, to musi być zadeklarowana przed wywołaniem. Postać tej deklaracji jest następująca:

typ nazwa_funkcji();

Lista argumentów nie musi być podawana. Reguły widoczności deklaracji funkcji są takie same jak w przypadku definicji zmiennej.

PRZYKŁAD

```
main(argc,argv)
int argc;
char **argv;
{
    FILE *fp, *fopen();            /* pierwszy wiersz */
    if ((fp = fopen(*argv, "r") = = NULL) /* drugi wiersz */
    ...
```

Pierwszy wiersz zawiera deklarację zmiennej fp o typie wskaźnikowym do struktury o nazwie FILE (struktura ta jest zdefiniowana w pliku stdio.h) i deklarację funkcji fopen, która daje w wyniku wskaźnik do tej struktury.

Gdyby brakowało tych deklaracji funkcji, to kompilator zaznaczyłby błędy w drugim wierszu programu, gdyż przyjąłby, że funkcja fopen daje w wyniku liczbę całkowitą int, a przypisanie liczby całkowitej wskaźnikowi jest niedopuszczalne.

PRZYKŁAD

Deklaracja	Znaczenie
int f()	f jest funkcją dającą w wyniku liczbę całkowitą
int *f()	f jest funkcją dającą w wyniku wskaźnik do liczby całkowitej

int (*)f()	f jest wskaźnikiem do funkcji dającej w wyniku liczbę całkowitą
int (*f[4])()	f jest tablicą 4 wskaźników do funkcji dających w wyniku liczby całkowite

PATRZ TAKŻE

wskaźnik do funkcji

---

deklaracja zmiennej

Deklaracja zmiennej nie powoduje zarezerwowania miejsca dla tej zmiennej, ale raczej wskazuje kompilatorowi, że zmienna jest zdefiniowana gdzie indziej. Zmienne są deklarowane także przez deklaracje extern.

PATRZ TAKŻE

deklaracja extern

---

do ... while

Składnia tej instrukcji jest następująca:

do *instrukcja* while *wyrażenie*

Najpierw jest wykonywana *instrukcja*, a potem ta sama *instrukcja* jest wykonywana tak długo, jak długo *wyrazenie* jest prawdziwe, to jest ma wartość niezerową.

Instrukcja do .. while jest zbliżona do instrukcji while, z tą różnicą, że w pierwszym przypadku *instrukcja* jest wykonywana co najmniej raz, w drugim natomiast może nie być wykonana ani razu, jeśli w wyniku obliczenia *wyrazenia* otrzyma się wynik zerowy. W przypadku instrukcji do .. while *wyrazenie* jest obliczane dopiero po pierwszym wykonaniu *instrukcji*.

Oczywiście *instrukcja* może być instrukcją złożoną.

PRZYKŁAD

```
do
   {
      func_jeden(i++);
      func_dwa(j++);
   } while (j <k);
```

Umieszczenie while w tym samym wierszu co klamra zamykająca ma służyć uniknięciu przez programistę pomyłki i uznaniu tej instrukcji za początek nowej instrukcji while. Wymagany na końcu znak średnika jest następnym przypomnieniem o jaką instrukcję chodzi.

---

efekty uboczne patrz *makrodefinicja*

---

enum

Zmienna typu wyliczeniowego – enum – ma ograniczony zakres, który musi być pokazany w jej definicji. Definicja zmiennej typu enum ma postać:

```
enum oznacznik
   {
      lista wartości
   }  nazwa zmiennej;
```

Nazwy wartości w *liście wartości* powinny być oddzielone przecinkami i nadaje się im wartości 0, 1, 2 itd.

Zmienne typu enum występują nie we wszystkich realizacjach kompilatorów języka C; w tych wersjach, w których nie występują, słowo *enum* nie jest słowem kluczowym.

PRZYKŁAD

Definicja:

```
enum tydzien
   {
      Niedziela;
      Poniedzialek
      Wtorek
      Sroda,
      Czwartek,
      Piatek,
      Sobota
   } dnitygodnia;
```

definiuje zmienną typu enum o nazwie dnitygodnia, której można przypisać wartości od Niedziela do Sobota, np.

```
dnitygodnia = Wtorek;
```

co nadaje tej zmiennej wartość 2. Wartość takiej zmiennej może być porównywana z wartościami dopuszczalnymi, np.

```
if (dnitygodnia = = Piatek)
```

Prostsze w użyciu niż zmienne typu enum są stałe symboliczne (patrz dyrektywa preprocesora #define).

EOF

Stała symboliczna EOF (koniec pliku) jest zdefiniowana w pliku stdio.h. Liczne funkcje standardowe dają w wyniku wartość EOF, oznaczającą koniec pliku lub inną nietypową sytuację.

Z powodów historycznych wartość EOF jest różna w różnych komputerach. Programista nie powinien czynić żadnych założeń na temat tej wartości i używać zawsze stałej symbolicznej zamiast na przykład –1.

---

etykieta

Etykieta nadaje instrukcji nazwę. Jej składnia jest następująca:

etykieta: *instrukcja*

Etykiety są używane w połączeniu z instrukcjami goto i switch. Zamiast *instrukcji* można użyć pary nawiasów klamrowych {}.

PRZYKŁAD

```
    if (k)
        goto problem;
    .
    .
    .
problem: error_routine(k);
    switch (i)
     {
        case 2 :   func(i,j);
                   break;
        case 4 :   func(j,i);
                   break;
        default :  func(j,j);
                   break;
     }
```

W drugim przykładzie (instrukcji switch) etykietami są case 2, case 4 oraz default i gdyby nie pośrednie instrukcje break, to zostałyby wykonane kolejno wszystkie instrukcje od func(i,j) do func(j,j).

PATRZ TAKŻE

goto
switch

---

for

Instrukcja for ma następującą składnię:

for (*wyrażenie1*;*wyrażenie2*;*wyrażenie3*) *instrukcja*

Wykonanie tej instrukcji odbywa się w taki sposób, że najpierw jest obliczane *wyrażenie1*. Potem powtarza się wykonanie *instrukcji* – pod warunkiem, że *wyrażenie2* jest prawdziwe (tzn. jego obliczenie daje wynik niezerowy), a następnie obliczanie *wyrażenia3*.

Każde z tych wyrażeń może zawierać wielokrotne wyrażenia, oddzielone przecinkami. Jest to najpopularniejszy przykład użycia operatora przecinkowego.

Każde z tych wyrażeń może być puste, ale nie można pominąć średników. Jeżeli *wyrażenie2* jest puste, to uważa się, że jest zawsze prawdziwe i instrukcja for staje się pętlą bez końca, którą można przerwać tylko za pomocą instrukcji return lub break (instrukcja continue nie powoduje wyjścia z pętli).

Instrukcja for jest równoważna następującej instrukcji while:

wyrażenie1;
while *wyrażenie2*
{
instrukcja
wyrażenie3;
}

Proste pętle nie wymagają *instrukcji* - może ona być pusta; wystarczają same wyrażenia.

PRZYKŁAD

```
char c[LINE_SIZE];
for (i = 0; i <LINE_SIZE; i+ +)
    c[i] = '\0';
for(i = 0; i <MAXLINES; line[i+ +] = 0)
    ;       /* instrukcja pusta, trzeba jednak wstawić średnik */
for(i = 0, j = 10; i, k; i + = j) /* wielokrotne inicjowanie */
```

Przykładem zwartości kodu w języku C może być poniższy program, który przekształca liczbę całkowitą na jej postać bitową i drukuje ją; wszystko to wykonuje w jednym wierszu.

```
#inlude <stdio.h>
#define BITS_IN_BYTE 8
main(argc, argv)
int argc;
char *argv[];
{
    int i,    /* liczba całkowita do przekształcenia */
        j,    /* licznik pętli */
        k,    /* tutaj wysunięte bity */
        l;    /* wartość aktualnego bitu*/
    char *itoa(); /* deklaracja funkcji */
    char buf1[17], buf2[17]:   /*bufory dla funkcji itoa */
    for ((i = itoa(argv[1]),j = 0, l = 1,k = (i & 1));
        j < (sizeof(int) * BITS_IN_BYTE);
        (j+ +, l < < =1, k =((i > > =1) &1)))
        printf("%c%s %s",
          (k ? '\n' : ' '),
          (k ? itoa(j,buf1,10) : ""),
          (k ? itoa(l,buf2,10) : ""));
}
```

Program wywołany z parametrem, na przykład:

```
program 7
```

daje w wyniku:

```
0 1
1 2
2 4
```

Jeśli w bibliotece dołączanej do programu jest stosowana inna postać niezbyt standardowej funkcji itoa, to należy odpowiednio zmienić kod źródłowy.

---

funkcja

W języku C nie ma rozróżnienia między funkcjami a procedurami; są tylko funkcje.

Funkcje muszą występować na tym samym poziomie leksykalnym, to znaczy, że nie jest możliwe definiowanie funkcji wewnątrz funkcji.

Domyślnie przyjmuje się, że wszystkie funkcje dają w wyniku wartość typu int. Funkcje dające w wyniku wartość innego typu muszą być deklarowane przed użyciem.

W niektórych wersjach C jest dopuszczony nowy typ void, używany tylko w definicji funkcji. Funkcja, której definicja przewiduje wynik typu void, faktycznie nie daje żadnego wyniku i użycie w niej instrukcji return(*wyrazenie*) jest błędem, podobnie jak błędem jest próba użycia gdziekolwiek w programie wyniku przekazywanego przez tak zdefiniowaną funkcję.

Wszystkie funkcje we wszystkich plikach źródłowych tworzących program i we wszystkich dołączanych bibliotekach są dla siebie jednakowo widoczne. Jedynym wyjątkiem jest funkcja zdefiniowana jako static, której zasięg jest ograniczony do pliku źródłowego, w którym została zdefiniowana.

Reguły zasięgu deklaracji funkcji są takie same jak reguły zasięgu deklaracji zmiennych.

Argumenty są przekazywane przez wartość, nie przez nazwę, z wyjątkiem tablic (ale nie elementów tablic), dla których jest przekazywany adres, i z wyjątkiem struktur (tylko w tych wersjach C, w których jest dopuszczalne użycie struktury jako argumentu funkcji), dla których także jest przekazywany adres.

Funkcja może zmieniać tylko zmienne lokalne albo zmienne zdefiniowane lub zadeklarowane jako globalne w pliku źródłowym, w którym jest ona zdefiniowana. Funkcja nie może zmienić zawartości zmiennej przekazanej do niej jako argument.

W celu zmodyfikowania niewidocznej dla niej zmiennej, funkcja musi mieć dostęp do adresu tej zmiennej albo dzięki temu, że adres jest zdefiniowany (lub zadeklarowany) globalnie albo dlatego, że został przekazany do funkcji jako argument.

Składnia definicji funkcji jest następująca:

klasa_pamięci typ_wyniku nazwa_funkcji(*lista_argumentów_formalnych*)
lista_argumentów_deklarowanych
{
definicje i deklaracje zmiennych
instrukcje
}

przy czym:

- *klasa_pamięci* może być extern (domyślna), static lub void;
- *typ_wyniku* może być jednym z podstawowych typów danych lub wskaźnikiem do tych typów, lub void; może być pominięty jeśli jest on int;
- *nazwa_funkcji* może być dowolną nazwą (identyfikatorem) dopuszczalną w C;
- *lista_argumentów_formalnych* powinna być zamknięta w nawiasach okrągłych; za nią znajduje się *lista_argumentów_deklarowanych*; liczba i nazwy argumentów w obydwu listach muszą być takie same, chociaż ich kolejność nie jest istotna.

PRZYKŁAD

```
static *int func_jeden(i, j, d, c)
double d;
char c;
int i, j;
{
    int q;
    q = 1;
    return (d * q);
}
```

Funkcja func_jeden jest klasy static (jej zasięg jest ograniczony do pliku źródłowego, w którym jest zdefiniowana); daje w wyniku wartość typu int; ma cztery argumenty (i, j, d, c). Zauważmy, że zmienne w każdej liście są wymieniane w innej kolejności, a zmienne i, j są zadeklarowane w tej samej instrukcji.

PATRZ TAKŻE

deklaracja funkcji
wywołanie przez nazwę i przez wartość
argument formalny i aktualny
wskaźnik do funkcji

---

goto

Instrukcja goto służy do przekazania sterowania programem bezpośrednio do innej instrukcji w tym samym bloku lub w obejmującym go bloku (ale nie do instrukcji poza funkcją).

Składnia instrukcji goto jest następująca:

goto *etykieta*;

przy czym *etykieta* jest "nazwą" innej instrukcji.

Stosowanie instrukcji goto jest powszechnie odradzane, gdyż jej nadmierne użycie prowadzi do błędów i trudności w zrozumieniu działania programu. Nie należy używać tej instrukcji, jeśli da się osiągnąć cel za pomocą instrukcji while, do .. while lub switch. Jej użycie zamiast return mogłoby być uzasadnione argumentacją, że instrukcja return jest niczym innym jak odmianą instrukcji goto, ale mimo to użycie goto jest odradzane. Użycie instrukcji goto może być uzasadnione tylko tam, gdzie może doprowadzić do istotnego uproszczenia zawikłanej struktury programu.

PRZYKŁAD

```
while (j--)
   {
      for (i = 0; j <MAX_ELEMENTS; i++)
         {
            if (count[i] <0)
               goto wczesny_koniec;
            func(count[i]);
         }
   }
wczesny_koniec: return;
```

Ponieważ instrukcja break spowodowałaby zakończenie pętli for, ale nie pętli while, więc wydaje się, że jedynym wyjściem jest użycie instrukcji goto. Jednak wielu programistów napisałoby pętlę while tak, żeby uniknąć stosowania instrukcji goto.

```
while (j--)
   {
      for (i = 0; j <MAX_ELEMENTS; i++)
         {
            if (count[i] <0)
               {
                  j = 0;            /* zakończenie pętli while */
                  break
               }
            else
                  func(count[i];
         }
   }
return;
```

PATRZ TAKŻE

etykieta

---

identyfikator

Identyfikator jest to nazwa utworzona z dowolnej kombinacji 26 liter alfabetu angielskiego (zarówno małych, jak i wielkich) i dziesięciu cyfr, przy czym nie może zaczynać się od cyfry. Ponadto dopuszcza się użycie podkreślenia w dowolnym miejscu identyfikatora, chociaż niektóre kompilatory (zwłaszcza te zgodne ze standardem ANSI) nie dopuszczają podkreślenia na początku identyfikatora.

Litery małe i wielkie są różne; na przykład abc i Abc są to różne identyfikatory.

W niektórych kompilatorach nakłada się ograniczenie na dopuszczalną długość identyfikatora, w innych nie ma takich ograniczeń.

Jeśli nawet kompilator nie nakłada ograniczeń na długość identyfikatora, to na ogół konsolidator nakłada ograniczenia na identyfikatory zewnętrzne.

PATRZ TAKŻE

rozdział 1 – informacje o identyfikatorach

---

identyfikator zewnętrzny

Identyfikator zewnętrzny jest znany wszystkim funkcjom.

Nazwy zmiennych zdefiniowanych między funkcjami i nazwami wszystkich funkcji, nie zdefiniowanych jako static, są znane zewnętrznie, to jest są znane konsolidatorowi.

Tak więc funkcje nie będące klasy static mogą być wywoływane z dowolnego pliku źródłowego. Zmienna, której nazwa jest znana zewnętrznie, może być deklarowana jako extern w innym pliku źródłowym.

PATRZ TAKŻE

deklaracja extern

---

idiom

Idiom jest to wyrażenie, które często powtarza się w programie. Oto przykłady idiomów:

```
while((c = getchar()) != EOF)
if ((fp = fopen(*++argv, "r") == NULL)
for(i = 0; i <MAX; i++)
```

W miarę jak programista zdobywa doświadczenie w programowaniu w C, te idiomy stają się częścią jego standardowego słownika. Regularne stosowanie idiomów prowadzi do właściwego stylu w programowaniu i oszczędza trudu "wymyślania koła" od nowa.

---

if .. else

W języku C występują dwie postaci instrukcji if .. else:

if (*wyrażenie*) *instrukcja1*

if (*wyrażenie*) *instrukcja1* else *instrukcja2*

W pierwszej postaci *instrukcja1* jest wykonywana, jeśli wartość *wyrażenia* nie jest zerowa.

W drugiej postaci *instrukcja1* jest wykonywana, jeśli wartość *wyrażenia* nie jest zerowa; przy zerowej wartości *wyrażenia* jest wykonywana *instrukcja2*.

Zarówno *instrukcja1*, jak i *instrukcja2* mogą być instrukcjami złożonymi.

Wątpliwości należy rozstrzygać łącząc każde else z ostatnim if nie mającym else.

Jeżeli *wyrażenie* jest złożone, tzn. zawiera operatory && i ||, to programista powinien użyć nawiasów do określenia kolejności obliczania. Brak takich nawiasów jest częstą przyczyną błędów.

PRZYKŁADY

```
if (i = = 5)
   q = 6;
if (i = = 4 || j >6)
   j = 2;
else
   j = 3;
if ((i = = 5 && j >6) || (k <2))
   {
      j = 2;
      k = 4;
   }
else
   k = 7;
if (i < =7)        /* pytanie co programista miał na myśli */
   if (q >4)
      j = 6;
else
   k = 4;
```

Zauważmy, że w tym ostatnim przykładzie słowo else – mimo przesunięcia wizualnego w wydruku – jest połączone z ostatnim if. Powinien on był napisać:

```
if (i < =7)
   {
      if (q >4)
         j = 6;
   }
else
   k = 4;
```

Pomijanie średnika przed słowem else jest częstym błędem programistów znających Pascal, ale początkujących w C.

Inicjowanie

Definicja zmiennej może także zawierać nadanie jej wartości początkowej (inicjowanie). Składnia jest następująca:

dla zmiennych prostych

definicja = *wyrażenie*;

dla agregatów (struktur i tablic)

definicja = {*lista_wartości_początkowych*};

Unie i agregaty klasy auto nie mogą być inicjowane.

Wyrażenie musi być stałą lub wynikiem jego obliczenia musi być stała albo adres poprzednio zadeklarowanej zmiennej (ewentualnie przesunięty o stałą wartość).

Lista_wartości_początkowych jest listą wartości dla elementów agregatu, oddzielonych przecinkami, w kolejności rosnących numerów tych elementów.

Zmienne klasy auto lub register mogą dodatkowo być inicjowane za pomocą wyrażeń zawierających wywołanie funkcji lub za pomocą poprzednio zdefiniowanych lub zadeklarowanych zmiennych, gdyż inicjuje się je podczas wykonywania programu, a nie podczas jego kompilacji.

W przypadku inicjowania agregatu, jeśli liczba elementów w liście wartości początkowych jest mniejsza niż liczba elementów agregatu, to pozostałym elementom agregatu nadaje się wartość zero.

Tablicy znakowej można nadać wartość początkową za pomocą ciągu znaków.

PRZYKŁAD

Instrukcja

```
int i = 3;
```

definiuje zmienną całkowitą i nadaje jej wartość początkową 3.

Instrukcja

```
int j[4] = {0,1,2,3};
```

definiuje tablicę 4 liczb całkowitych i nadaje im następujące wartości:

```
j[0] = 0
j[1] = 1
j[2] = 2
j[3] = 3
```

Instrukcja

```
int k[] = {1,2,3};
```

definiuje tablicę 3 liczb całkowitych (liczba elementów tablicy jest określana na podstawie liczby elementów listy wartości początkowych) i nadaje im następujące wartości:

```
k[0] = 1
k[1] = 2
k[2] = 3
```

Instrukcja

```
int m[4] = {1,2};
```

definiuje tablicę 4 liczb całkowitych i nadaje im następujące wartości:

```
m[0] = 0
m[1] = 1
```

Reszcie elementów tablicy, dla których nie zostały podane wartości początkowe, nadaje się wartość zero.

Instrukcja

```
int n[4][3] = {{1,2,3}, {4,5,6}};
```

definiuje dwuwymiarową tablicę (4 wiersze i 3 kolumny) liczb całkowitych i nadaje im następujące wartości:

```
n[0][0] = 1
n[0][1] = 2
n[0][2] = 3
n[1][0] = 4
n[1][1] = 5
n[1][2] = 6
```

Pozostałym elementom tablicy (wiersze n[2] i n[3]) nadaje się wartość zero.
Instrukcja

```
int n[4][3] = {1,2,3,4,5,6};
```

definiuje tablicę taką samą jak w poprzednim przykładzie i nadaje jej takie same wartości.
Instrukcja

```
int q[4][3] = {{1},{2},{3},{4}};
```

definiuje tablicę dwuwymiarową i nadaje jej elementom następujące wartości:

```
q[0][0] = 1
q[1][0] = 2
q[2][0] = 3
q[3][0] = 4
```

Pozostałym elementom tablicy nadaje się wartość zero. Instrukcja

```
char *test = "To jest przykład";
```

definiuje ciąg znaków tak duży, aby zmieścił się w nim tekst, stanowiący jego wartość początkową, łącznie z kończącym każdy ciąg znaków znakiem '\0'.

Zauważmy, że dwa ciągi znaków, nawet jeśli nadano im tę samą wartość początkową, są różne.
Instrukcja

```
struct
  {
     char *word;
     int count;
  } start[4] = { "hello", 0, "goodbye", 0 };
```

definiuje tablicę 4 struktur, przy czym pierwszym dwom elementom tablicy (strukturom) nadaje się następujące wartości początkowe:

```
start[0].word = "hello"
start[0].count = 0
start[1].word = "goodbye"
start[1].count = 0
```

Pozostałym elementom struktury nadaje się wartość zero.

---

instrukcja

Wyrażenie zakończone średnikiem jest uważane za instrukcję. Na przykład

```
i+1
```

jest wyrażeniem, podczas gdy

```
i+1;
```

jest instrukcją.

Oto przykłady innych instukcji:

- ; instukcja pusta
- if (*wyrażenie*) *instrukcja* – patrz *if*;
- if (*wyrażenie*) *instrukcja1* else *instrukcja2* – patrz *if*;
- while (*wyrażenie*) *instrukcja* – patrz *while*;
- do *instrukcja* while (*wyrażenie*); – patrz *do .. while*;
- for (*wyrażenie1;wyrażenie2;wyrażenie3*) *instrukcja* – patrz *for*;
- switch *wyrażenie_całkowitoliczbowe*

```
{
    case stała1: instrukcja1
    case stała2: instrukcja2
    .
    .
}
```

patrz *switch*

- break; – patrz *break*;
- continue; – patrz *continue*;
- goto *etykieta*; – patrz *goto*;
- return; – patrz *return*;
- return *wyrażenie*; – patrz *return*.

---

instrukcja złożona

Instrukcja złożona, zwana także blokiem, jest to zamknięty w nawiasy klamrowe ciąg instrukcji, z których część lub wszystkie mogą być instrukcjami złożonymi. Instrukcja złożona może być użyta w każdym miejscu, w którym w składni języka jest dopuszczone użycie instrukcji.

Przed pierwszą instrukcją prostą instrukcji złożonej mogą znajdować się definicje i deklaracje zmiennych i deklaracje funkcji.

PRZYKŁAD

```
if (x == y)
    a = b;
else
    {
      int j = 0;
```

```
        j = b;
        b = c;
    }
```

---

klasa pamięci

Zmienne mogą należeć do jednej z czterech klas pamięci.

1. Klasa static

Jeśli zmienna jest zdefiniowana poza funkcjami (to jest między definicjami funkcji) jako zmienna klasy static, to można się do niej odwołać z dowolnego miejsca pliku źródłowego. Jeżeli taka zmienna jest zdefiniowana wewnątrz funkcji lub bloku, to można się do niej odwołać tylko z wnętrza tej funkcji lub bloku.

W każdym z przypadków zmienną klasy static można inicjować tylko za pomocą wyrażenia dającego w wyniku stałą. Inicjowanie zmiennych klasy static – nawet tych, które zostały zdefiniowane wewnątrz funkcji lub bloków – odbywa się na początku wykonywania programu.

2. Klasa extern

Zmienna klasy extern jest zdefiniowana zewnętrznie, to jest poza plikiem źródłowym lub funkcją, w których jest zadeklarowana jako extern.

Gdy zmienna jest zadeklarowana jako extern poza funkcją, to można się do niej odwołać z dowolnego miejsca pliku źródłowego. Jeżeli taka zmienna jest zdefiniowana wewnątrz funkcji lub bloku, to można się do niej odwołać tylko z wnętrza tej funkcji lub bloku.

Zmienna extern nie może być inicjowana, gdyż kompilator nie wie, gdzie ona się znajduje. Odwołania do zmiennych extern łączy konsolidator.

3. Klasa auto

Zmienna klasy auto (automatyczna) jest to taka zmienna, która została zdefiniowana wewnątrz funkcji lub bloku. Za każdym razem, gdy blok lub funkcja zaczyna być wykonywana, przydziela się jej pamięć, a po zakończeniu działania bloku lub funkcji pamięć ta jest zwalniana.

Do zmiennej klasy auto można się odwołać tylko wewnątrz funkcji lub bloku, w którym została zdefiniowana. Można jej nadać wartość początkową za pomocą dowolnego wyrażenia, gdyż operacja ta odbywa się w czasie pracy programu.

4. Klasa register

Zmienna klasy register jest zapamiętywana – o ile tylko jest to możliwe – w rejestrze komputera. Ponieważ architektury komputerów różnią się znacznie między sobą, więc nie ma żadnej gwarancji, że zmienna klasy register będzie istotnie zapamiętana w rejestrze komputera.

Zmienną klasy register można definiować tylko wewnątrz funkcji lub bloku i można się do niej tylko tam odwoływać.

Gdy pominięto klasę pamięci przy definiowaniu zmiennej, to nadaje się jej domyślnie klasę auto, jeśli jest definiowana wewnątrz funkcji lub bloku, i klasę static w przeciwnym razie. Jeśli taka zmienna została zdefiniowana poza funkcjami (to jest między definicjami funkcji), to można się do niej odwołać z dowolnego miejsca pliku

źródłowego lub z innego pliku źródłowego, w którym została zdeklarowana jako extern.

PATRZ TAKŻE

zasięg
czas życia zmiennej

---

kolejność obliczeń

Język C nie gwarantuje, że argumenty w wyrażeniu będą obliczane w jakiejś ustalonej kolejności. Na przykład we fragmencie

```
i = 0;
func(+ +i, i + 2);
```

wartość aktualnych argumentów przekazanych do funkcji func może równie dobrze być 1 i 3, jak i 1 i 2; nie ma co do tego żadnych reguł. Inny przykład niejasności:

```
j = 0;
i[j] = j+ +;
```

W tym przypadku 0 lub 1 może być przypisane i[0] lub i[1].

```
i = func_jeden() – func_dwa();
```

Także tu nie ma żadnej pewności, która funkcja będzie wywołana pierwsza.

Dostępny w systemie operacyjnym Unix program lint wskazuje takie wątpliwe miejsca.

---

komentarz

Komentarz jest ograniczony znakami /* na początku i */ na końcu. Nie ma żadnych ograniczeń co do rozmiarów komentarza ani liczby jego wierszy. Komentarze są ignorowane przez kompilator; preprocesor usuwa je przed fazą właściwej kompilacji.

Nie można definiować komentarza zagnieżdżonego, to jest wewnątrz innego komentarza. Jeśli coś takiego się zdarzy, to zestaw znaków /* otwierający drugi komentarz będzie traktowany jako część pierwszego i zostanie zignorowany. Pierwsze pojawienie się zestawu */ spowoduje zamknięcie pierwszego komentarza i kompilator będzie uważał wszystko to, co następuje za zestawem */ jako tekst programu C, co prawdopodobnie doprowadzi do licznych błędów składniowych.

Dość powszechny błąd polega na tym, że zestaw */ kończący komentarz jest pisany niepoprawnie, na przykład jako * / (ze spacją między znakami). Wówczas cały tekst między pierwszym zestawem /* a następnym poprawnie napisanym zestawem końcowym */ jest uważany za komentarz. Jeśli nie ma następnego poprawnego zestawu kończącego, to cała reszta tekstu jest traktowana jako komentarz. Mimo to może się zdarzyć, że kompilacja nie wykaże błędów.

PRZYKŁAD

```
int i = 0; /* to jest poprawnie zakończony komentarz */
int j = 0; /* to jest niepoprawnie zakończony komentarz * /
```

```
int k = 0; /* k nie jest zdefiniowane, mimo pozorów poprawności */
```

PRZYKŁAD

```
if (q = = 5)
    j + + ; /* ten program nie robi tego, czego oczekiwał * /
else
    j– –; /* programista (brak w nim instrukcji "else") */
```

---

kompilacja rozłączna

Program w języku C składa się z plików tekstowych, które są kompilowane do postaci pośredniej (obiektowej). Te pliki pośrednie są łączone z innymi plikami pośrednimi oraz z biblioteką (lub bibliotekami) w celu utworzenia pliku wykonywalnego.

Te dwa kroki (kompilacja i konsolidacja) są wymagane nawet dla najprostszych programów, gdyż wiele właściwości języka zrealizowano w postaci funkcji bibliotecznych, a nie w postaci instrukcji języka C.

Tworzenie programu z zestawu małych plików zmniejsza czas kompilacji, gdyż nie zmienione pliki nie muszą być kompilowane powtórnie. Ponadto małe pliki łatwiej się redaguje.

W systemie operacyjnym Unix (i w niektórych innych systemach) przyjęto następującą umowę co nazw plików:
- pliki źródłowe mają rozszerzenie .c
- pliki zawierające tyko definicje i deklaracje mają rozszerzenie .h
- pliki pośrednie mają rozszerzenie .o
- pliki wykonywalne nie mają rozszerzenia.

PRZYKŁAD

Program o nazwie korelacja może składać się z następujących plików:
- korelacja.c – zawiera funkcje;
- korelacja.h – zawiera definicje danych i deklaracje;
- korelacja.o – jest plikiem pośrednim, wejściowym dla konsolidatora;
- korelacja – jest plikiem wykonywalnym.

Jeśli plik korelacja.c staje się zbyt duży (z punktu widzenia autora lub edytora tekstowego), to można go podzielić na dwa pliki, na przykład korelacja1.c i korelacja2.c. Będą wtedy istniały także dwa pliki pośrednie korelacja1.o i korelacja2.o, powstałe w wyniku oddzielnej kompilacji plików źródłowych. Zmiana w pliku korelacja1.c będzie wymagała skompilowania tylko tego pliku; pliku korelacja2.c nie trzeba będzie kompilować. Konsolidator połączy te dwa pliki z biblioteką i utworzy z nich plik korelacja.

Pliki o rozszerzeniu .h są włączane do kompilowanego tekstu za pomocą dyrektywy preprocesora #include. Plik wejściowy dla kompilatora, złożony z pojedynczego pliku o rozszerzeniu .c i wszystkich włączonych plików o rozszerzeniu .h, nosi nazwę pliku źródłowego; z kompilatora otrzymujemy pojedynczy plik pośredni (o rozszerzeniu .o). Niektóre kompilatory dopuszczają wywołanie kompilatora z podaniem większej liczby plików o rozszerzeniu .c w wierszu poleceń, ale w tym przy-

padku każdy z nich jest kompilowany w oddzielnym przebiegu kompilatora; po każdym takim przebiegu powstaje jeden plik o rozszerzeniu .o dla każdego pliku o rozszerzeniu .c.

Plik źródłowy nie musi odpowiadać plikowi fizycznemu. Plik źródłowy jest to obiekt logiczny złożony z pojedynczego pliku o rozszerzeniu .c i wszystkich włączonych plików o rozszerzeniu .h, z wyłączeniem tych instrukcji, które zostały usunięte przez dyrektywy preprocesora.

---

konsolidator

Konsolidator jest to program łączący jeden lub więcej plików wyjściowych z kompilatora z podanymi bibliotekami i tworzący wykonywalny program wynikowy.

Pliki wyjściowe z kompilatora nie mogą być wykonywane, ponieważ zawierają odwołania do funkcji bibliotecznych (minimum to podprogramy inicjowania oraz funkcje wejścia-wyjścia), a także odwołania do funkcji lub zmiennych zdefiniowanych w innych plikach źródłowych. Zadanie konsolidatora polega na połączeniu tych odwołań.

Składnia dyrektyw konsolidatora zależy od realizacji i od systemu operacyjnego. Jednym z często denerwujących aspektów użycia konsolidatora jest kolejność wymieniania bibliotek. Na przykład konsolidator z systemu operacyjnego Unix przegląda biblioteki dwukrotnie; jeśli jakaś funkcja z biblioteki odwołuje się do funkcji w innej bibliotece, to ta druga biblioteka musi być wymieniona jako druga. Czasami wzajemne powiązania funkcji są tak skomplikowane, że biblioteki trzeba wymieniać więcej niż jeden raz. W takim przypadku korzystnie jest połączyć kilka małych bibliotek w jedną większą.

PATRZ TAKŻE

kompilacja rozłączna

---

konwersja

W wyrażeniach arytmetycznych z operatorami dwuargumentowymi, takimi jak mnożenie lub dodawanie, jeśli argumenty są różnego typu, to kompilator musi wygenerować odpowiednie instrukcje przekształcające je do typów wzajemnie zgodnych, według następującej procedury.

1. Argumenty typu float są przekształcane do typu double, a argumenty typu char lub short do typu int.

2. Jeśli jeden argument jest typu double, to drugi też jest przekształcany do typu double, wynik jest typu double i następuje wyjście z procedury konwersji.

3. Jeśli jeden argument jest typu long, to drugi też jest przekształcany do typu long, wynik jest typu long i następuje wyjście z procedury konwersji.

4. Jeśli jeden argument jest typu unsigned, to drugi też jest przekształcany do typu unsigned, wynik jest typu unsigned i następuje wyjście z procedury konwersji.

5. Po osiąnięciu tego punktu oba argumenty muszą być typu int i wynik też jest typu int.

W wyrażeniach złożonych ta procedura jest stosowana tak długo, aż całe wyrażenie zostanie obliczone. W wyrażeniach przypisania, wartość wyrażenia z prawej strony jest przekształcana do typu odpowiadającego lewemu argumentowi, to jest do typu wyniku.

Gdy zmienna short int jest przekształcana do typu int, to jest zachowany znak.

Gdy zmienna char jest przekształcana do int, to rozszerzenie znaku może następować lub nie – zależnie od charakterystyki sprzętu komputera.

Gdy zmienna int jest przekształcana do char, to nadmiarowe najbardziej znaczące bity są gubione.

Gdy zmienna long jest przekształcana do char, short lub int, to nadmiarowe najbardziej znaczące bity są gubione.

Gdy zmienna float jest przekształcana do int, to jest gubiona część ułamkowa. Pewne cechy tej konwersji zależą od właściwości komputera.

Zmienna double jest przekształcana do typu float przez zaokrąglanie.

Reguły konwersji zachowują "bezznakowość" liczby, w przeciwieństwie do reguł "zachowania wartości", zdefiniowanych w proponowanym standardzie ANSI.

Przy wywołaniu funkcji także następuje konwersja argumentów zgodnie z następującymi regułami.

Zmienne typu char i short są przekształcane do typu int.

Zmienne float są przekształcane do typu double.

Zauważmy, że te reguły konwersji argumentów w wywołaniu funkcji powodują, że aktualny argument typu char będzie poprawnie przekazany zamiast argumentu formalnego typu int, ale nie typu long. Na ogół złą praktyką programowania jest poleganie na automatycznej konwersji argumentów w wywołaniu funkcji; programista powinien użyć odpowiedniego typu lub skorzystać z możliwości konwersji typów.

PATRZ TAKŻE

konwersja typów

---

konwersja typów

Operator konwersji typów (ang. *cast*) wymusza konwersję między typami danych, która bez jego użycia nie zostałaby wykonana. Operator konwersji typów, użyty na przykład w wywołaniu funkcji, pozwala uniknąć definiowania typu wyników pośrednich.

PRZYKŁAD

Przyjmijmy, że są podane następujące definicje:

```
int i; /* definicja zmiennej */
func(l) /* definicja funkcji */
long l;
{
}
```

Tak zdefiniowaną funkcję można wywołać z argumentem i, jeśli na zmiennej i zostanie dokonana konwersja typu, jak poniżej:

```
func((long)i);
```

---

l-wartość

L-wartość jest to obiekt podlegający przetwarzaniu, czyli zmienna, której można przypisać wartość, co ją odróżnia od stałej. Nazwa l-wartość wynikła z faktu, że jest to "lewostronna wartość', czyli obiekt, który może pojawić się po lewej stronie instrukcji przypisania.

Do l-wartości można stosować jednoargumentowy operator &.

Nazwa tablicy jest rozumiana przez kompilator jako adres jej piewszego elementu, czyli stała, a nie l-wartość.

PRZYKŁAD

```
int n[4][3];
```

definiuje dwuwymiarową tablicę (4 wiersze i 3 kolumny).

Nazwa n nie jest l-wartością, ale stałą, czyli adresem n[0][0]; sam element n[0][0] jest l-wartością. Podobnie n[0] nie jest l-wartością, ale stałą, czyli adresem pierwszego wiersza tablicy.

Operacje &n i &n[0] są niedopuszczalne.

łączność

Wyrażenie złożone, czyli zawierające więcej niż jeden operator, jest obliczane zgodnie z regułami priorytetu i łączności operatorów.

Jeśli w wyrażeniu wszystkie operatory mają ten sam priorytet, to są one stosowane do argumentów od prawej do lewej lub od lewej do prawej, zależnie od rodzaju łączności w danej grupie operatorów. Na przykład wyrażenie

```
1 + 2 + 3
```

jest obliczane jako ((1 + 2) + 3) ponieważ operator + jest łączny lewostronnie. Tak się szczęśliwie składa, że wynik tego obliczenia jest taki sam, jak wynik obliczenia (1 + (2 + 3)).

W wielu przypadkach, np. jeśli w tym samym wyrażeniu występuje przesuwanie w lewo i w prawo, różna łączość operatorów może dać różne wyniki. Chcąc zadać określoną kolejność obliczania wyrażeń, należy używać nawiasów.

---

main

Każdy program musi mieć dokładnie jedną funkcję o nazwie main. Pierwsza wykonywalna instrukcja w programie wynikowym odpowiada pierwszej instrukcji w programie źródłowym w funkcji main. Funkcja main nie musi być pierwszą funkcją w programie źródłowym.

W wielu implementacjach funkcja main może akceptować dwa argumenty, dostarczane przez system operacyjny: argc – będący liczbą argumentów w wierszu

poleceń przy wywoływaniu programu (łącznie z nazwą programu), oraz argv – wskaźnik do tablicy ciągów znaków, zawierających argumenty z wiersza poleceń. W niektórych implementacjach przewidziano dodatkowy argument – wskaźnik do struktury zawierającej informacje o środowisku.

Argument argc ma wartość co najmniej równą 1, gdyż nazwa programu (zawsze obecna) jest uważana za pierwszy argument.

PRZYKŁAD

```
main(argc,argv)
int argc;
char *argv[];
{
   FILE *fp, *fopen();
   if (argc != 3)
    {
      printf("\nPotrzebne są dwa argumenty");
      exit();
    }
   else
    {
      if((fp = fopen(argv[1],"r")) == NULL)
        {
          printf("\nNie mogę otworzyć pliku %s",argv[1]);
          exit();
        }
      else
        process_file(fp);
    }
}
```

Argument argv można także deklarować jako **argv.

---

makrodefinicja

Makrodedfinicja jest to szybka metoda zapisywania często używanych instrukcji. Niektóre z prostszych funkcji bibliotecznych są w rzeczywistości realizowane jako makrodefinicje. W wielu kompilatorach przewidziano zarówno kompilowaną realizację niektórych funkcji, jak i wersje w postaci makrodefinicji.

Makrodefinicję podaje się za pomocą dyrektywy preprocesora #define i od tej chwili może ona być swobodnie używana w programie. Składnia makrodefinicji jest następująca:

#define *nazwa_makro* (*lista_argumentów*)

Między nazwą makrodefinicji a nawiasem otwierającym listę argumentów nie może być spacji. Jeśli będzie spacja, to preprocesor uzna, że #define jest definicją stałej symbolicznej.

Jeśli w makrodefinicji występuje więcej niż jedna instrukcja, to muszą one być zawarte w nawiasach klamrowych. Jeśli trzeba kontynuować makrodefinicję w kolejnym wierszu, to wiersz poprzedni należy zakończyć znakiem ukośnika \.

PRZYKŁAD
Instrukcja

```
#define max(a,b) (((a) > (b)) ? (a) : (b))
```

definiuje makro o nazwie max rozwijające się do wyrażenia, którego wartość jest równa większemu z argumentów. Można by uważać, że makrodefinicja działa jak funkcja, ale występują jednak pewne różnice.

Makrodefinicja max może być użyta w następujący sposób:

```
i = max(i,j)
```

Tak zdefiniowane makro ma dwa poważne niedociągnięcia.

1. Efekty uboczne

Rozważmy następujący fragment programu:

```
i = 5;
j = 2;
i = max(++i,j);
```

Na pierwszy rzut oka wydaje się, że zmiennej i zostanie przypisana liczba 6. W rzeczywistości powyższe wyrażenie rozwija się do:

```
i = ((++i) > (j)) ? (++i) : (j))
```

przy czym ++i jest obliczane dwa razy i zmiennej i zostanie przypisana liczba 7. Te nieoczekiwane efekty wielokrotnego obliczania wyrażeń w makrodefinicji są znane pod nazwą efektów ubocznych i występują jedynie w makrodefinicjach. Jeśli max byłoby funkcją, to taki efekt nie wystąpiłby, gdyż argumenty funkcji są obliczane tylko raz, a poza tym funkcja ma dostęp tylko do kopii argumentów.

Efektów ubocznych można uniknąć przez staranne dobieranie argumentów do makrodefinicji, czyli przez używanie tylko argumentów "prostych", bez operatorów i wywołań funkcji.

2. Kolejność wykonywania

Zauważmy, że w makrodefinicji występują nawiasy wokół każdego argumentu rozwinięcia. Jest to niezbędne, ponieważ brak nawiasów mógłby spowodować obliczenie rozwinięcia w nieoczekiwany sposób, wynikający z priorytetu operatorów. W definicji

```
#define cube(i) (i * i * i)
```

wyrażenie cube(j + 2) rozwinie się do (j + 2 * j + 2 * j + 2), zamiast do ((j + 2) * (j + 2) * (j + 2)), z tego powodu, że operator * ma wyższy priorytet niż operator +. Poprawny sposób zdefiniowana tego makro jest następujący:

```
#define cube(i) ((i) * (i) * (i))
```

Używanie makrodefinicji daje oszczędności związane z brakiem wywołania funkcji, ale ponieważ makrodefinicja rozwija się do swej pełnej postaci tyle razy, ile razy została użyta, więc program wynikowy może być większy niż powinien być.

PRZYKŁAD

```
#define lo_byte(w) ((w) & 0xff)
#define hi_byte(w) lo_byte((w) >> 8)
```

Zauważmy, że to drugie makro zostało zdefiniowane za pomocą pierwszego.

Uruchamianie makrodefinicji jest trudne, gdyż teksty programu w edytorze i kompilatorze są różne. W tym przypadku programista powinien wywołać oddzielnie preprocesor i sprawdzić jego produkt wyjściowy.

PRZYKŁAD

```
#define DUZE_MAKRO(a,b) {func((a)); \
                         func((b));}
```

NULL

NULL jest to stała symboliczna zdefiniowana w pliku stdio.h, a służąca do wskazywania niepoprawnej wartości wskaźnika. Kompilator ostrzeże przed próbą przypisania stałej wartości wskaźnikowi lub porównania wskaźnika ze stałą, chyba że tą stałą jest NULL.

Nie dla wszystkich komputerów NULL ma wartość 0 i dlatego programiści powinni używać raczej NULL niż 0.

Błędem jest przyjmowanie, że NULL wskazuje komórkę pamięci, której zawartość jest zerowa.

Niektóre z funkcji przydziału pamięci dają w wyniku NULL (por. dodatek A) w razie pojawienia się błędu.

---

operator

W języku C występuje 45 operatorów w 15 grupach. W każdej z tych grup operatory mają ten sam priorytet. Grupy operatorów są wyliczone w malejącej kolejności priorytetów.

Operator jednoargumentowy akceptuje tylko jeden argument, operator dwuargumentowy – dwa, a operator trójargumentowy akceptuje trzy argumenty.

Dla każej grupy wymieniono także zasady łączności operatorów.

1. Operatory łączne lewostronnie

()	wywołanie funkcji	getc(stdin)

Kompilator nie sprawdza, czy parametry aktualne odpowiadają parametrom formalnym ani co liczby, ani co do typu.

[]	odwołanie do elementu tablicy	i[17]

Nie sprawdza się, czy indeks tablicy mieści się w granicach określonych w jej definicji.

->	wskaźnik do elementu struktury	alt_ptr->c

Nie sprawdza się, czy wskaźnik istotnie wskazuje element struktury.

odwołanie do elementu struktury alt.c

2. Operatory łączne prawostronnie

–	minus jednoargumnetowy	–i

Nie należy tego operatora mylić z operatorem dwuargumentowym, oznaczającym odejmowanie.

+ +	inkrementacja (zwiększenie o 1)	i+ +

Ten operator może mieć dwie postaci: przedrostkową i przyrostkową, czyli może być umieszczony przed lub po zmiennej, której dotyczy. Jeśli jest napisany przed zmienną (+ +i), to zmienna jest zwiększana przed obliczeniem wyrażenia; jeśli zaś po zmiennej, to jest ona zwiększana po obliczeniu wyrażenia. Jest to o tyle użyteczny operator, że często jest kompilowany bezpośrednio na instrukcję inkrementacji wykonywaną sprzętowo.

– –	dekrementacja (zmniejszenie o 1)	i– –

Ten operator może mieć dwie postaci: przedrostkową i przyrostkową, czyli może być umieszczony przed lub po zmiennej, której dotyczy. Jeśli jest napisany przed zmienną (– –i), to zmienna jest zmniejszana przed obliczeniem wyrażenia; jeśli zaś po zmiennej, to jest ona zmniejszana po obliczeniu wyrażenia. Jest to o tyle użyteczny operator, że często jest kompilowany bezpośrednio na instrukcję dekrementacji wykonywaną sprzętowo.

!	negacja logiczna	!znaleziony
~	uzupełnienie do 1	~0x7f
*	wskazanie pośrednie	*c_ptr

Nie należy mylić operatora * z operatorem dwuargumentowym, oznaczającym mnożenie; *c_ptr oznacza obiekt wskazywany przez c_ptr, będący domyślnie wskaźnikiem.

&	adres elementu	&i

Nie należy tego operatora mylić z operatorem dwuargumentowym, oznaczającym bitowy iloczyn logiczny.

Jednoargumentowy operator & nie może być stosowany do zmiennych klasy register ani do pól bitowych.

sizeof(*obiekt*)	rozmiar zmiennej lub typu	sizeof(int)

Operator sizeof ma postać wywołania funkcji, ale w rzeczywistości jest operatorem i daje w wyniku rozmiar obiektu w bajtach. Obiektem może być nazwa zmiennej lub typ, takie jak int lub element struktury.

(*type*)	zmiana typu	(int)c

Ten operator dokonuje konwersji wyrażenia do podanego typu.

3. Operatory łączne lewostronnie

*	mnożenie	i * j

Nie należy mylić tego operatora z jednoargumentowym operatorem *, oznaczającym "obiekt wskazywany przez".

/	dzielenie	i / j
%	operacja modulo	i % j

Operatora modulo % nie można stosować do zmiennych typu float lub double. Przykładowo, 5 % 2 daje w wyniku 1, czyli resztę z dzielenia.

4. Operatory łączne lewostronnie

+	dodawanie	i + j
–	odejmowanie	i – j

Nie należy mylić operatora – z operatorem jednoargumentowym, oznaczającym zmianę znaku.

5. Operatory łączne lewostronnie

< <	przesunięcie bitowe w lewo	i < < 2

Wyrażenie i < < 2 oznacza przesunięcie w lewo o 2 bity. Brakujące bity są uzupełniane zerami.

> >	przesunięcie bitowe w prawo	i > > 4

Wyrażenie i > > 4 oznacza przesunięcie w prawo o 4 bity. Brakujące bity są uzupełniane zerami, jeśli argument jest typu unsigned. W niektórych implementacjach przesunięcie może być arytmetyczne (z zachowaniem znaku).

6. Operatory łączne lewostronnie

<	mniejszy niż	i < j
< =	mniejszy lub równy	i < = 6
>	większy niż	i > 2
> =	większy lub równy	i > = j

Te operatory dają w wyniku 0, jeśli relacja jest fałszywa, i 1 jeśli jest prawdziwa.

7. Operatory łączne lewostronnie

= =	równy	if(i = = 5)
! =	nierówny	if(i ! = 5)

Te operatory dają w wyniku 0, jeśli relacja jest fałszywa, i 1 jeśli jest prawdziwa.

8. Operatory łączne lewostronnie

&	bitowy iloczyn logiczny	c & 033

Dwuargumentowego operatora & nie należy mylić z jednoargumentowym operatorem &, oznaczającym adres obiektu.
Dwuargumentowy operator & może być stosowany jedynie do liczb całkowitych.

9. Operatory łączne lewostronnie

^	bitowa suma modulo 2	c ^ 0317

Dwuargumentowy operator ^ może być stosowany jedynie do liczb całkowitych.

10. Operatory łączne lewostronnie

|	bitowa suma logiczna	c | 0333

Dwuargumentowy operator | może być stosowany jedynie do liczb całkowitych.

11. Operatory łączne lewostronnie

&&	iloczyn logiczny	i = = 5 && j = = 6

Ten operator daje w wyniku 1 (prawda), jeśli obydwa argumenty są niezerowe (prawdziwe), i zero (fałsz) jeśli chociaż jeden z argumentów jest zerowy.

12. Operatory łączne lewostronnie

|| suma logiczna i == 5 || j == 6

Ten operator daje w wyniku 1 (prawda), jeśli chociaż jeden z argumentów jest niezerowy, i 0 gdy obydwa argumenty są zerowe. Obliczanie odbywa się od lewej strony do prawej i nie oblicza się drugiego argumentu, jeśli pierwszy jest niezerowy.

13. Operatory łączne prawostronnie

? : wyrażenie warunkowe i > 4 ? i : j

Składnia tego trójargumentowego operatora jest następująca:

wyrażenie1 ? *wyrażenie2* : *wyrażenie3*

Najpierw jest obliczane *wyrażenie1*. Jeśli jest ono niezerowe, to w wyniku otrzymuje się *wyrażenie2*, w przeciwnym razie *wyrażenie3*. Ten operator jest dogodnym skrótem dla często używanej konstrukcji. Zamiast pisać:

```
if (i < 4)
    k = 1;
else
    k = 2;
```

można napisać:

```
k = (i > 4 ? 1 : 2);
```

14. Operatory łączne prawostronnie

=	przypisanie	i = 7
*=	mnożenie, potem przypisanie	i *= 3
/=	dzielenie, potem przypisanie	i /= 4
%=	modulo, potem przypisanie	i %= 4

Operator %= nie może być stosowany do argumentów typu float lub double.

+=	dodawanie, potem przypisanie	i += 5
-=	odejmowanie, potem przypisanie	i -= 6
&=	iloczyn bitowy, potem przypisanie	i &= 0333
^=	suma modulo 2, potem przypisanie	i ^= 0317
|=	iloczyn logiczny, potem przypisanie	i |= 0177
<<=	przesunięcie w lewo, potem przypisanie	i <<= 2
>>=	przesunięcie w prawo, potem przypisanie	i >>= 3

Operatory w tej grupie (z wyjątkiem =) dają alternatywną metodę wykonywania operacji i przypisania wyniku jednemuo z argumentów. Na przykład i = i + 1 może być napisane jako i += 1, a i = i >> 2 może być napisane jako i >>= 2. Ta krótsza postać pozwala uniknąć błędów takich jak napisanie i = j + 2, gdy miało się na myśli i = i + 2.

Typ operacji przypisania jest taki, jak typ argumentu po lewej stronie operatora.

15. Operatory łączne lewostronnie
obliczenie i odrzucenie   i = 5, j

Składnia tego operatora jest następująca:

wyrażenie1, wyrażenie2

Wyrażenie1 jest obliczane i wynik tego obliczenia odrzucany; następnie jest obliczane *wyrażenie2* i wynik jest wynikiem operacji. Jeśli operator ten jest stosowany w liście argumentów w wywołaniu funkcji, to może pojawić się tylko w nawiasach. Na przykład

```
(i = 2, i + 3)
```

da w wyniku 5,

```
func(j, (i = 2, i + 4), k)
```

powoduje wywołanie funkcji z aktualnymi argumentami j, 6, k. Gdyby nie było wewnętrznych nawiasów w wywołaniu funkcji, to byłyby cztery argumenty aktualne: j, 2, 4, k.

---

plik źródłowy patrz *kompilacja rozłączna*

---

pliki standardowe

W systemie operacyjnym Unix, gdy program zaczyna się wykonywać, następuje automatyczne otwarcie trzech plików. Są to (w nawiasach podano wskaźniki tych plików):

- standardowe wejście (stdin)
- standardowe wyjście (stdout)
- standardowe wyjście błędów (stderr)

Przy korzystaniu z tych plików programista może używać buforowanych funkcji wejścia-wyjścia; pliki te są automatycznie przypisane do terminala, ale można je skierować do pliku lub potoku.

PRZYKŁAD

Instrukcja

```
put(c,stdout);
```

powoduje zapisanie znaku do pliku stdout.

---

pola bitowe

Pola bitowe są to poszczególne bity wewnątrz zmiennej będącej liczbą całkowitą (typu int). Pola bitowe nie mogą przekraczać granic liczby. Jeśli pole bitowe jest zbyt szerokie, aby zmieścić się do końca słowa, to zaczyna się na granicy nowego słowa.

Do pól bitowych nie można stosować jednoargumentowego operatora &.

Pole bitowe nie może mieć szerokości większej niż słowo. Pole bitowe bez nazwy oznacza przerwę w ciągu bitów słowa.

Aby wymusić rozpoczynanie się pola bitowego od początku słowa, należy zdefiniować przed nim pole bitowe bez nazwy o długości równej zero.

Kierunek przydzielania pól bitowych (od lewej do prawej lub odwrotnie) zależy od realizacji sprzętowej komputera. W celu zapewnienia przenośności oprogramowania, nie należy w programie czynić żadnych założeń dotyczących kierunku przydziału pól bitowych.

Mimo braku w definicji języka C założeń stwierdzających, że pola bitowe mają być typu unsigned, w większości implementacji kompilatorów tak właśnie się je realizuje.

PRZYKŁAD
Przy założeniu, że komputer jest 16-bitowy, zdefiniujmy strukturę:

```
struct
   {
    unsigned field_a : 3 ;
    unsigned field_b : 11 ;
    unsigned field_c : 4 ;
    unsigned : 0 ;
    unsigned field_d : 4;
   } pole_bitowe ;
```

Tak zdefiniowane pole_bitowe zajmuje trzy kolejne słowa. Pole field_c zaczyna się od nowego słowa, gdyż pola field_a i field_b zajmują razem 14 z 16 bitów słowa, co nie pozostawia dość miejsca na pole field_c. Pole bitowe field_d zaczyna się od trzeciego słowa, gdyż pole bez nazwy, o długości zero, znajdujące się przed nim, wymusza ustawienie pola field_d od początku następnego słowa, mimo że w słowie drugim zostało jeszcze dostatecznie wiele miejsca, aby mogło się tam zmieścić pole bitowe field_d. Pozostałe bity trzeciego słowa są nieużywane.

---

preprocesor
Pierwszą fazę kompilacji wykonuje preprocesor, który podstawia tekst i warunkowo wyłącza część tekstu z kompilacji. W systemie opracyjnym Unix jest możliwe uruchomienie preprocesora oddzielnie; nie jest to cecha szczególna języka C i można ją spotkać także w innych językach programowania.

Preprocesor usuwa z tekstu źródłowego komentarze przed przekazaniem go do kompilatora.

Instrukcje preprocesora, zwane dyrektywami, zawsze zaczynają się od znaku # zaczynającego wiersz tekstu. Oto te dyrektywy:

1. #define; realizuje parametryzowane podstawianie tekstu; jest także używana do definiowania stałych symbolicznych i makro z parametrami;

2. #if *wyrażenie_stałe instrukcje* #endif; jeśli *wyrażenie_stałe* jest niezerowe, to włącza instrukcje aż do #endif do tekstu kompilowanego;

3. #if *wyrażenie_stałe instrukcje* #else *instrukcje* #endif; podobna do poprzedniej dyrektywy; zawiera instrukcje po #else;

4. #ifdef *identyfikator instrukcje* #endif; jeśli w tej części tekstu *identyfikator* jest zdefiniowany, to włącza instrukcje aż do #endif do tekstu kompilowanego;

5. #ifdef *identyfikator instrukcje* #else *instrukcje* #endif; podobna do poprzedniej dyrektywy; zawiera instrukcje po #else;

6. #ifndef *identyfikator instrukcje* #endif; jeśli w tej części tekstu *identyfikator* nie jest zdefiniowany, to włącza instrukcje aż do #endif do tekstu kompilowanego;

7. #ifndef *identyfikator instrukcje* #else *instrukcje* #endif; podobna do poprzedniej dyrektywy; zawiera instrukcje po #else;

8. #include "*nazwa_pliku*" lub #include <*nazwa_pliku*>; włącza ten plik do tekstu kompilowanego;

9. #line *n* "*plik*"; identyfikuje następne wiersze jako pochodzące z "*pliku*", wiersz numer *n*;

10. #undef *identyfikator*; powoduje, że definicja *identyfikatora* nie jest już rozpoznawana.

---

priorytet operatorów

Priorytet operatorów określa kolejność obliczania skomplikowanego wyrażenia. Na przykład, wyrażenie

```
1 + 2 * 3
```

przy obliczaniu może dać wynik 9 (jeśli najpierw wykona się dodawanie) lub 7 (jeśli najpierw wykona się mnożenie). Z każdym z 45 operatorów języka C jest związany priorytet i wyrażenia są obliczane zgodnie z hierarchią priorytetów. Ponieważ mnożenie ma wyższy priorytet niż dodawanie, więc powyższe wyrażenie da w wyniku 7.

Jeśli programista życzy sobie, aby wyrażenie było obliczone w inny sposób, to może określić kolejność wykonywania przez użycie nawiasów. Na przykład (1 + 2) * 3 daje w wyniku 9.

Jeśli ptr jest wskaźnikiem do struktury, której elementem jest i, to wyrażenie ++ptr->i spowoduje inkrementację i, a nie wskaźnika ptr, gdyż operator -> ma wyższy priorytet niż ++. W celu wykonania inkrementacji wskaźnika programista musi napisać (++ptr)->i, ustanawiając właściwe priorytety.

W haśle operatory znajduje się kompletna lista operatorów i ich priorytetów.

PATRZ TAKŻE

operatory
łączność

---

przenośność

Język C nie gwarantuje przenośności oprogramowania, ale ponieważ definicja języka nie jest zależna od żadnego konkretnego komputera, więc programy napisane w C można łatwo przenosić. Program napisany dla jednego komputera może być niewielkim wysiłkiem przeniesiony na inny komputer, wszakże pod pewnymi warunkami.

1. Obydwa kompilatory muszą być standardowe lub rozsądnie bliskie przyjętym standardom. Istnieje wiele implementacji mikrokomputerowych kompilatorów

języka z niestandardowymi funkcjami wejścia i wyjścia. Programiści powinni unikać tych kompilatorów.

2. W programie nie można czynić założeń co do właściwości sprzętu komputera. Te właściwości to ustawienie obiektów, rozmiary danych różnych typów i kolejność zapamiętywania bitów w liczbie całkowitej.

Jeśli konkretny komputer wymaga, aby obiekty określonego typu były ustawione na granicy słowa lub liczby całkowitej, to w strukturach realizujących te wymagania mogą pojawić się "dziury"; tych "dziur" może nie być przy kompilowaniu programu w innym komputerze. Tak więc programista nie powinien czynić nawet najbardziej elementarnych założeń na temat względnego położenia elementów struktury.

Błędem jest przyjmowanie, że obiekty jednego typu mają taki sam rozmiar lub są większe niż obiekty drugiego typu. Należy po prostu unikać mieszania typów. Ponadto, jeśli jest potrzebna wiedza na temat rozmiaru określonego typu, na przykład przy użyciu liczb całkowitych do odwzorowania pól bitowych, to należy używać operatra sizeof.

Niektóre z komputerów zapamiętują bajty liczby całkowitej w odwróconej kolejności, to jest bity mniej znaczące mają adres mniejszy niż bity bardziej znaczące. Nie należy więc także czynić założeń co do kolejności zapamiętywania bajtów słowa.

3. Różne kompilatory dopuszczają różne maksymalne długości identyfikatorów. Identyfikatory zewnętrzne muszą być zadawane zgodnie z wymaganiami konsolidatora. Dobrze jest, jeśli identyfikatory mają niewielką długość; w każdym razie powinny dać się jednoznacznie odróżnić na podstawie 7–8 znaków. Fakt, że program daje się skompilować na nowym komputerze nie oznacza, że będzie pracował tak samo.

Jeśli na przykład jeden kompilator sprawdza wszystkie znaki identyfikatora przy określaniu jego znaczenia, a drugi tylko 8 znaków, to fragment programu

```
i = process_last_node_not_reached();
j = process_last_node_reached();
```

skompiluje się różnie w tych komputerach. Drugi kompilator wywoła dwa razy tę samą funkcję. Jeśli te funkcje znajdują się w bibliotece, to konsolidator dołączy jedną lub drugą. Być może nawet nikt nie zauważy, że przy umieszczaniu w bibliotece jedna zapisała się na drugiej. Taka sytuacja będzie bardzo trudna do diagnozowania.

4. Dla różnych implementacji bardzo prawdopodobne są różnice w największej długości ciągu znaków lub innego obiektu.

5. Niestaranność przy przekazywaniu argumentów może przejść nie zauważona na jednym komputerze, ale spowoduje kłopoty na drugim. Dla większości komputerów 32–bitowych typy long i int są identyczne i mieszanie ich nie szkodzi. Jeśli jednak program jest kompilowany na komputer 16–bitowy, na którym long ma 32 bity, a int 16 bitów, to natychmiast powstają problemy. Zauważmy, że także stałe mają określane typy; należy więc stosować stałe typu long.

6. Przyjmowanie, że wskaźniki są równoważne liczbom całkowitym albo że wskaźnik do jednego typu danych jest taki sam jak do innego typu, jest poważnym błędem podważającym przenośność oprogramowania.

7. W programie nie można przyjmować założeń na temat kolejności obliczania lub rodzaju efektów ubocznych, gdyż zależą one od implementacji, a nawet od wersji tej samej implementacji.

8. W programie nie należy używać zmiennej typu char przy wykonywaniu operacji arytmetycznych.

9. W programie nie można przyjmować założenia, że bit najbardziej znaczący oznacza znak liczby, ale należy tę liczbę porównać z zerem.

---

rekurencja

Rekurencja jest to technika programowania polegająca na tym, że funkcja może wywołać samą siebie (rekurencja bezpośrednia) lub wywołać inną funkcję, która z kolei wywołuje tę pierwszą (rekurencja pośrednia). Rekurencja dobrze nadaje się do rozwiązywania problemów lub przetwarzania danych, które zostały zdefiniowane rekurencyjnie, takich jak problem wieży z Hanoi lub drzew binarnych.

W języku C, gdy funkcja zaczyna się wykonywać, są tworzone jej lokalne zmienne automatyczne. Przy rekurencyjnym wywołaniu funkcji jest tworzony drugi zestaw tych zmiennych, ale nadal istnieją te pierwsze. W miarę pogłębiania rekurencji rośnie zapotrzebowanie na pamięć. To szybko rosnące zapotrzebowanie na pamięć jest najpoważniejszym kosztem rekurencji.

Utworzona lokalnie w funkcji zmienna klasy static nie jest tworzona na nowo przy każdym wywołaniu funkcji. Kolejne wywołania funkcji odwołują się do tej samej zmiennej.

PRZYKŁAD

```
func(i)
int i;
{
    int j = 0;
    static int k;
    j++;
    k++;
    func(j);
}
```

Niezależnie od tego, ile razy zostanie wywołana funkcja func, istnieje tylko jedna zmienna k, ale za każdym wywołaniem jest tworzona na nowo zmienna j.

---

return

Instrukcja return kończy wykonanie funkcji, w której występuje.

Istnieją dwie postaci instrukcji return:

return;

return *wyrażenie*;

Pierwsza postać po prostu kończy wykonanie funkcji, druga natomiast powoduje przekazanie wartości do wywołującej instrukcji. Nie ma potrzeby zamykania *wyrażenia* w nawiasy.

Kompilator wskazuje jako błędną każdą instrukcję znajdującą się za bezwarunkową instrukcją return, gdyż do tej instrukcji program nigdy nie dojdzie. Druga postać instrukcji return nie może być używana w funkcjach zdefiniowanych jako void.

PRZYKŁAD

```
return i + 5;
i++;
```

Instrukcja return jest poprawnie użyta, ale następująca po niej instrukcja (i++;) nigdy nie zostanie osiągnięta i kompilator wskaże ją jako błąd składniowy. To samo jest prawdziwe dla poniższego przykładu.

```
if (i ==20)
  {
   return;
   i++;
  }
```

Zauważmy, że kompilator nie wskaże jako błędu użycia instrukcji po bezwarunkowym wywołaniu funkcji exit lub _exit, mimo że ta instrukcja także nigdy nie będzie wykonana; kompilator nie "rozumie", że wywołanie tej funkcji kończy wykonanie programu. Zresztą, nawet gdyby kompilator miał taką informację, to nic nie może stanąć na przeszkodzie temu, aby programista zdefniował własną funkcję exit lub _exit, wykonującą całkiem inne działania niż funkcja z biblioteki, która nie zostanie wtedy użyta.

PATRZ TAKŻE

opis funkcji exit i _exit w dodatku A

---

rozszerzenie znaku

Zwykle zmiennych typu char używa się do pamiętania wartości ze standardowego zestawu znaków komputera, na ogół ASCII lub EBCDIC. Są one na pewno nieujemne.

Jeżeli ujemna wartość zmiennej typu char zostanie przekształcona na typ int, to ta liczba int może być różna dla różnych komputerów – ujemna lub dodatnia. Ta niejednoznaczność nie ma znaczenia dla znaków ze standardowego zestawu znaków, o których wiadomo na pewno, że są nieujemne, ale dla innych wartości zmiennej typu char może prowadzić do programów nieprzenośnych. Z tego powodu należy używać zmiennych unsigned char.

---

sekwencje specjalne

Znak ukośnika \ oznacza początek sekwencji specjalnej, to jest ciągu znaków o specjalnym znaczeniu:

\n   nowy wiersz (LF)

\t	tabulator poziomy
\b	cofacz
\r	powrót karetki (CR)
\f	nowa stronica (FF)
\\	ukośnik
\'	apostrof
\ddd	wzór bitowy, w którym ddd jest sekwencją od 1 do 3 liczb ósemkowych

W ciągu znaków zestaw \" reprezentuje cudzysłów.

---

słowa kluczowe

Następujące słowa, zwane słowami kluczowymi, są zastrzeżone, gdyż mają specjalne znaczenie dla kompilatora języka C i nie mogą być używane jako identyfikatory (nazwy zmiennych, funkcji lub etykiety):

auto	else	int	typedef
break	entry	long	switch
case	enum	register	union
char	extern	return	unsigned
continue	float	short	void
default	for	sizeof	while
do	goto	static	
double	if	struct	

Słowa entry i enum nie są słowami kluczowymi w niektórych implementacjach C. W niektórych implementacjach słowa asm i fortran są słowami kluczowymi.

W standardzie ANSI zdefiniowano trzy dodatkowe słowa kluczowe: const, signed i volatile.

---

sprawdzanie typów

W języku C nie przestrzega się ściśle sprawdzania typów danych. Argumenty o różnych typach są przekształcane zgodnie z regułami podanymi w haśle konwersja typów.

Większość kompilatorów, mimo że ostrzega o przypisaniu wskaźnikowi liczby całkowitej i odwrotnie, to jednak kompiluje te instrukcje. W wielu komputerach wskaźniki i liczby całkowite są identyczne, co pozwala swobodnie mieszać te typy. Takie użycie nie jest jednak przenośne i powinno być unikane.

Błędem jest przyjmowanie, że wskaźnik do jednego typu danych jest taki sam jak wskaźnik do innego typu. Jedynie stałą 0 można porównywać ze wskaźnikiem lub przypisywać wskaźnikowi; ze względu na przenośność należy w tym miejscu stosować stałą symboliczną NULL zamiast 0.

---

stała

Stała wartość musi być określonego typu, podobnie jak zmienne mogą być tylko określonych typów; może ona być użyta wszędzie tam, gdzie dopuszcza się użycie zmiennej. Stałe w programie podaje się w następujący sposób:

Typ	Składnia stałej	Przykład
char	ujęta w apostrofy	'a'
ciąg znaków	ujęta w cudzysłowy	"abc"
int	nie może mieć 0 na początku	123
liczba ósemkowa	na początku musi mieć 0	0377
liczba szesnastkowa	na początku musi miec 0x	0x2f
long int	kończy się literą l lub L	123L
float	zawiera kropkę dziesiętną lub	3.24 lub
	pisana w notacji naukowej	3.2E4
	(patrz niżej)	
double	jak float	

Wszystkie stałe typu float są uważane za typ double. Przez notację naukową rozumie się następującą sekwencję znaków:

część całkowita (może być ze znakiem)
kropka dziesiętna
część ułamkowa
litera e lub E
wykładnik całkowity (może być ze znakiem)

Część całkowita i część ułamkowa są pisane jako ciąg cyfr. Można pominąć kropkę dzisiętną lub literę e (E), ale nie można pominąć obydwu tych znaków. Podobnie, można pominąć część ułamkową lub część całkowitą, ale nie można pominąć obydwu tych części.

PRZYKŁAD

```
if (c = = '\n')
i = '\007'
l = 0L
f = 34e-22
```

Stałe mogą być użyte w następujących miejscach, gdzie nie dopuszcza się stosowania zmiennych.

1. W definicji tablicy:

```
int *p_i[10];
```

Stała służy w tym miejscu do określenia liczby elementów tablicy.

2. W definicji pola bitowego:

```
struct
   {
      unsigned field_a : 3;
      unsigned field_b : 11;
      unsigned field_c : 4;
      unsigned  : 0;
      unsigned field_d : 4;
   } bit_field ;
```

Stałe służą w tym miejscu do określenia szerokości pola (w bitach).

3. Po słowie kluczowym case w instrukcji switch.

We wszystkich tych przypadkach można także stosować wyrażenia, których obliczenie da w wyniku stałą, pod warunkiem, że zawiera ono tylko stałe (takie jak 2 + 2) i nie zawiera zmiennych ani wywołania funkcji, gdyż jest obliczane podczas kompilacji.

---

stałe symboliczne

Stałą symboliczną definiuje się za pomocą dyrekytwy #define preprocesora. Zaleca się stosowanie stałych symbolicznych zamiast zwykłych stałych, ponieważ w razie konieczności zmiany stałej wystarczy zmienić jej wartość w jednym miejscu – w dyrektywie #define.

PRZYKŁAD

```
#define MAX_LINE_SIZE 128
char line[MAX_LINE_SIZE];
```

Zwykle w programie jest wiele miejsc, w których występuje liczba znaków w zmiennej line, na przykład zmienna line może mieć nadaną wartość początkową za pomocą następującej instrukcji:

```
for (i = 0; i <MAX_LINE_SIZE; i+ +)
   line[i] = '\0';
```

Jeśli, zamiast używać stałej symbolicznej MAX_LINE_SIZE programista użyje po prostu stałej 128, to późniejsza decyzja zmiany rozmiaru tablicy oznacza konieczność dokonania zmiany w (co najmniej) dwóch instrukcjach, z katastrofalnymi wynikami, gdy gdzieś zapomni się dokonać tej zmiany.

Przy wyborze nazw stałych symbolicznych należy zadbać o to, aby nie pojawiły się nazwy już użyte, zwłaszcza użyte przez system. Jeśli podejrzewamy, że może wystąpić taki konflikt nazw, to należy uruchomić preprocesor oddzielnie i sprawdzić uzyskane wyniki (tylko w tych implementacjach, w których jest to możliwe).

Przyjęło się pisać nazwy stałych symbolicznych wielkimi literami, tak jak w powyższych przykładach, dzięki czemu wyróżniają się one w programie.

---

standard ANSI języka C

Komitet X3J11 Amerykańskiego Narodowego Instytutu Standardów (ANSI) opublikował w październiku 1986 roku propozycję standardu języka C. Ogólnie rzecz biorąc, w standardzie wprowadzono tylko nieznaczne zmiany do języka C, ale za to bardziej rygorystycznie zdefiniowano pewne konstrukcje językowe. Programy, które nie zależą od słabo zdefiniowanych aspektów języka i zostały napisane z myślą o tym, aby były niezależne sprzętu, mają duże szanse być zgodne z proponowanym standardem. Oto najważniejsze cechy standardu.

1. Zdefiniowano trzy nowe słowa kluczowe: const, signed i volatile.

2. Identyfikatory zaczynające się od podkreślenia zostały zarezerwowane dla nazw zewnetrznych, takich jak funkcje biblioteczne.

3. Funkcje konwersji arytmetycznych zostały tak zdefiniowane, aby zachować wartość, a nie "bezznakowość", jak to zaproponowali Kernighan i Ritchie.

4. Wprowadzono drobne zmiany do dyrektywy #define.

5. Wiele funkcji operujących na znakach, takich jak isalpha, powinno dobrze działać także przy użyciu różnych alfabetów narodowych.

Dodatkowe informacje na temat standardu można otrzymać z:
CBEMA
311 First Street NW
Suite 5000
Washington, DC 2001
USA

---

stderr patrz *pliki standardowe*

---

stdin patrz *pliki standardowe*

---

stdio.h

Plik stdio.h zawiera definicje struktur i stałych symbolicznych (na przykład EOF) używane przez biblioteczne buforowane funkcje wejścia-wyjścia. Zwykle trzeba włączyć (dyrektywą #include) ten plik do każdego programu korzystającego z tych funkcji.

Stałe NULL, TRUE i FALSE są także zdefiniowane w stdio.h.

stdout patrz *pliki standardowe*

---

struktura

Struktura (słowo kluczowe w postaci struct) jest to zbiór logicznie związanych zmiennych różnych typów, odpowiadający rekordowi w języku Pascal. Deklaracja struktury *oznacznik* jest z wielu punktów widzenia równoważna deklarowaniu nowego typu, to znaczy, że teraz można definiować zmienne o typie struct *oznacznik*, wskaźniki do struct *oznacznik* itd. Strukturę można określić na jeden z trzech sposobów.

1. struct *oznacznik* {*lista_deklaracji*};

Deklaracja struktury nazwanej *oznacznik*, bez rezerwowania dla niej pamięci i bez definiowania jakiejkolwiek zmiennej.

2. struct {*lista_deklaracji*} *nazwa_zmiennej*;

Deklaracja nie nazwanej struktury i zdefiniowanie pojedynczej zmiennej (lub tablicy zmiennych), których typ jest taki jak typ struktury.

3. struct *oznacznik* {*lista_deklaracji*} *nazwa_zmiennej*;

Kombinacja dwóch powyższych przypadków: deklaracja struktury i zdefiniowanie zmiennej o takim typie.

Jedyne operacje jakie można wykonywać na strukturze to znalezienie jej adresu (za pomocą operatora &) i odwołanie się do jej elementów (z użyciem operatora . lub kombinacji operatorów * i . albo ich odpowiednika ->). W niektórych kompilatorach jest możliwe przekazanie struktury jako argumentu funkcji.

PRZYKŁADY

```
struct complex
   {
      double real;
      double imaginary;
   };
```

Zadeklarowano tu strukturę o nazwie complex, ale nie zdefiniowano żadnej zmiennej typu struct complex. Definicje podane poniżej są poprawne pod warunkiem, że odbywają się w zasięgu powyższej deklaracji:

```
struct complex c1;
struct complex *c1;
struct complex *func();
struct complex c[2];
```

W pierwszej definiuje się zmienną typu struct complex, w drugiej – wskaźnik do typu struct complex; w trzeciej deklaruje się funkcję dającą w wyniku wskaźnik do struktury typu struct complex, a w ostatniej definiuje się tablicę dwóch zmiennych typu struct complex.

```
struct
   {
      double real;
      double imaginary;
   } c;
```

Zadeklarowano tu nie nazwaną strukturę i zdefiniowano pojedynczą zmienną tego typu. Ponieważ ta struktura nie ma oznacznika, nie jest więc możliwe zdefiniowane innych zmiennych lub wskaźników do struktury tego typu.

```
struct complex
   {
      double real;
      double imaginary;
   } c[2];
```

Zadeklarowano tu strukturę o nazwie complex i zdefiniowano tablicę dwóch zmiennych typu struct complex. Definicje i deklaracje podane dla pierwszego przykładu są także tutaj dopuszczalne, pod warunkiem, że znajdują się w zasięgu tej deklaracji.

Elementami struktury mogą być zmienne dowolnego typu, łącznie z polami bitowymi, uniami i innymi strukturami.

Wskaźnik do deklarowanej struktury może być zdefiniowany jako jeden z jej elementów. Taka właściwość nosi nazwę autoreferencji struktury.

PRZYKŁAD

```
struct node
   {
      struct node *parent;
      struct node *right_child;
      struct node *left_child;
      char c;
   };
```

Do poszczególnych elementów struktury można odwoływać się w różny sposób. Przy podanych poniżej deklaracjach i definicjach:

```
struct complex
   {
      float real;
      float imaginary;
   } c;

struct complex *p_c = c;
```

wyrażenie p_c jest wskaźnikiem do struktury c; wyrażenia c.real, (*p_c).real, p_c->real odnoszą się do zmiennej real w strukturze c.

Wyrażenie c.real dosłownie znaczy "zmienna real w strukturze c", natomiast (*p_c).real znaczy "zmienna real w strukturze wskazywanej przez p_c". Nawiasy są niezbędne, gdyż priorytet operatora . jest wyższy niż priorytet operatora *.

Wyrażenie p_c->real ma dokładnie to samo znaczenie co (*p_c).real, a ponieważ w tej postaci stosuje się tylko jeden operator, unika się więc kłopotów z priorytetami operatorów i wobec tego należy tę postać preferować. Operator -> składa się ze znaku łącznika –, po którym bez odstępu następuje znak większości > .

Ponieważ operatory . i -> znajdują się w grupie pierwszej operatorów o najwyższym priorytecie, trzeba więc uważać przy używaniu ich w połączeniu z innymi operatorami w wyrażeniach. Na przykład ++p_c->real spowoduje inkrementację zmiennej real, a nie p_c. W celu wykonania inkrementacji p_c przed pobraniem real należy napisać (++p_c)->real.

Przy odwołaniu do elementów struktury zakłada się, że wyrażenie po lewej stronie operatora . lub -> wskazuje strukturę, której elementem jest argument po prawej stronie operatora. W rzeczywistości to całkiem rozsądne wymaganie wcale nie jest wymuszane. Wyrażenie po lewej stronie operatora . może być dowolną l-wartością całkowitą, a wyrażenie po lewej stronie operatora -> może być liczbą całkowitą lub wskaźnikiem do dowolnej liczby całkowitej. Innymi słowy, z punktu widzenia kompilatora cokolwiek może wskazywać cokolwiek. Kompilator w żaden sposób nie sprawdza podczas kompilacji, czy programista używa właściwego wskaźnika; z całą pewnością też nie sprawdza się w czasie działania programu, czy wskaźnik ma "sensowną wartość". Zadaniem programisty jest zapewnienie, aby wskaźniki wskazywały właściwe obiekty.

Struktury nie można traktować jako jednostki. Kopiowanie struktury do innej struktury może się odbywać jedynie przez kopiowanie jej elementów indywidualnie. Zapis struktury do pliku odbywa się przez zapisanie każdego jej elementu lub bajtu oddzielnie.

PATRZ TAKŻE

pola bitowe

unie

Dodatek D – przykłady dynamicznego przydziału pamięci dla struktur

---

struktura programu

Program w języku C składa się z jednej lub więcej funkcji na tym samym poziomie leksykalnym. Nie ma takiego rozróżnienia między procedurą a funkcją, jakie występuje na przykład w Pascalu – tu są znane tylko funkcje. Nie można także definiować funkcji wewnątrz funkcji – wszystkie funkcje znajdują się na tym samym poziomie leksykalnym.

Zarówno zmienne, jak i funkcje muszą być zadeklarowane zanim mogą być użyte; wyjątkiem są funkcje dające w wyniku liczbę całkowitą, które nie muszą być deklarowane przed użyciem.

Każdy program musi mieć dokładnie jedną funkcję o nazwie main. Pierwsza wykonywalna instrukcja w programie wynikowym odpowiada pierwszej instrukcji w programie źródłowym w funkcji main.

Kolejność funkcji w pliku źródłowym nie ma znaczenia, pod warunkiem wszakże, że funkcje dające wynik inny niż liczba całkowita powinny być zadeklarowane przed ich pierwszym użyciem. Deklaracja funkcji nie określa jej argumentów, jedynie typ wyniku. Ponadto, kompilator języka C nie sprawdza czy lista argumentów aktualnych w wywołaniu funkcji odpowiada co do liczby i rodzaju argumentom formalnym.

Argumenty są przekazywane przez wartość, a nie przez nazwę. Funkcja nie może zmienić wartości argumentu, który został jej przekazany. Funkcja może zmienić wartość zmiennej zdefiniowanej lokalnie w innej funkcji tylko wówczas, gdy adres tej zmiennej znajduje się w jej zasięgu, tzn. został zdefiniowany globalnie albo został jej przekazany jako argument.

Zmienne zdefiniowane między definicjami funkcji są znane jako identyfikatory zewnętrzne i są znane konsolidatorowi.

PATRZ TAKŻE

zasięg

---

switch

Instrukcję switch stosuje się tam, gdzie należy dokonać wyboru na podstawie pojedynczego wyrażenia całkowitoliczbowego, o ograniczonym zakresie wartości. Składnia tej instrukcji jest nastepująca:

switch (*wyrażenie*) *instrukcja*

Wyrażenie musi dawać w wyniku liczbę całkowitą. *Instrukcja* jest zwykle instrukcją złożoną, w której poszczególne instrukcje są poprzedzone przedrostkiem postaci:

case *wyrażenie_stałe*:

Żadna para przedrostków nie może mieć tej samej wartości tego wyrażenia. Co najwyżej jeden przedrostek może mieć postać:

default:

Przy wykonywaniu instrukcji switch oblicza się *wyrażenie* i porównuje z wartością *wyrażenia_stałego* w przedrostkach. Jeśli są równe, to jest wykonywana pierwsza instrukcja za tym przedrostkiem, w przeciwnym razie jest wykonywana instrukcja za przedrostkiem default (jeśli występuje). Jeśli nie nastąpiła zgodność wartości *wyrażenia* i *wyrażenia_stałego* w którymkolwiek z przedrostków, a nie ma przedrostka default, to nie wykonuje się żadna instrukcja w instrukcji switch.

PRZYKŁAD

```
int i, j, k;
switch (i)
   {
      case 0 : k++;
        break;
      case 1 : j++;
        break
      default : j--;
        break;
   }
```

Gdy i jest równe 0, sterowanie przechodzi do instrukcji k++. Gdy i jest równe 1, sterowanie przechodzi do instrukcji j++. Gdy i nie jest równe 0 ani 1, sterowanie przechodzi do instrukcji j--.

Zauważmy, że przedrostki case są zwykłymi etykietami i w trakcie wykonywania są pomijane. W razie braku instrukcji break instrukcja j++ zostałaby wykonana po instrukcji k++. Zalecamy umieszczanie zawsze break po ostatniej instrukcji w instrukcji switch, zwłaszcza że nigdy nie wiadomo, czy nie zechcemy dodać więcej etykiet case trochę później.

---

tablica

Tablica jest zbiorem zmiennych tego samego typu, zapamiętywanych w kolejnych komórkach pamięci. Tablicę definiuje się następująco:

typ nazwa_zmiennej[*stała*] ...

Stała może być wyrażeniem, które po obliczeniu daje stałą, ale nie może zawierać zmiennych ani wywołań funkcji, gdyż jest obliczane w czasie kompilacji.

PRZYKŁAD

Instrukcja

```
int i[10];
```

definiuje tablicę 10 liczb całkowitych i[0], i[1],... i[9]. Zauważmy, że pierwszy element zawsze ma numer zero. W tym przypadku nie istnieje element i[10] i błędem jest odwoływanie się do niego, nawet jeśli kompilator nie wykaże takiego odniesienia jako błędu – co zresztą jest częstym powodem trudności w uruchamianiu. Instrukcja

```
int j[4][3];
```

definiuje tablicę 12 liczb całkowitych:

```
j[0][0], j[0][1], j[0][2]
j[1][0], j[1][1], j[1][2]
j[2][0], j[2][1], j[2][2]
j[3][0], j[3][1], j[3][2]
```

Tę tablicę możemy określić jako tablicę o 4 wierszach i 3 kolumnach.

Nazwa tablicy jest stałą mającą znaczenie adresu jej pierwszego elementu. W tablicach wielowymiarowych, jak tablica j powyżej, stała j jest adresem elementu j[0][0] (zmiennej). Stała j[0] jest adresem pierwszego elementu pierwszego wiersza tablicy, to jest j[0][0]. Stała j[1] jest adresem pierwszego elementu w drugim wierszu w tablicy, to jest j[1][0] itd.

Specjalnym rodzajem tablicy jest ciąg znaków, definiowany jako tablica znaków; na przykład:

```
char informacja[72];
```

definiuje ciąg znaków o długości 72 znaki.

PRZYKŁAD
Instrukcja

```
int *p[4];
```

definiuje tablicę 4 wskaźników do liczb całkowitych. Instrukcja

```
int (*p[4])();
```

definiuje tablicę 4 wskaźników do funkcji, dających w wyniku liczby całkowite.

PATRZ TAKŻE
inicjowanie

---

test na wartość niezerową
Test czy wartość nie jest zerowa jest domyślnie wykonywany przy każdym testowaniu i dlatego może być zawsze pominięty. Na przykład, równie dobrze można napisać if(i != 0), jak i if(i). Trzy poniższe instrukcje :

```
if(i)
if(i != 0)
if((i != 0) != 0)
```

mają dokładnie to samo znaczenie.

---

typy danych

W języku C podstawowe typy danych to:

Typ	Znaczenie	Typowy rozmiar (bitów)
char	pojedynczy znak	8
int	liczba całkowita	zależy od realizacji
short int	liczba całkowita krótka	16
long int	liczba całkowita długa	32
float	liczba zmiennopozycyjna	32
double	liczba zmiennopozycyjna o podwójnej precyzji	64
unsigned char	znak nieujemny	8
unsigned int	liczba całkowita nieujemna	16

Zakres liczb zmiennopozycyjnych o pojedynczej i podwójnej precyzji zwykle wynosi od –10E38 do 10E38.

Typy wymienione w powyższym zestawieniu są oznaczone słowami kluczowymi używanymi do definiowania lub deklarowania zmiennych, ale dopuszcza się skróty: short zamiast short int oraz long zamiast long int.

Inne typy danych to:

– tablica
– struktura
– unia
– wskaźniki do powyższych typów, łącznie ze wskaźnikami do wskaźników.

---

typedef

Deklaracja typedef służy do nadania nowej nazwy istniejącemu typowi danych. Przez użycie tej deklaracji nie tworzy się nowych typów danych. Nowa nazwa może być używana wszędzie tam, gdzie nazywany przez nią typ danych mógł być stosowany. Składnia tej deklaracji jest następująca:

typedef *typ_danych nowa_nazwa*;

PRZYKŁAD

```
typedef int INTEGER; /* zamiast int można teraz używać INTEGER */
typedef struct
    {
        float real;
        float imaginary;
    } COMPLEX;    /* typ danych o nazwie COMPLEX */
INTEGER i = 0;  /* odpowiada napisaniu int i = 0; */
COMPLEX c1,c2, *add_complex(); /* definiuje dwie struktury i deklaruje
        funkcję dającą w wyniku wskaźnik do struktury typu COMPLEX */
```

Użycie deklaracji typedef do nadania nowej nazwy strukturze zawierającej autoreferencję jest procesem dwustopniowym:

```
struct node
   {
      struct node *parent;
      struct node *right_child;
      struct node *left_child;
      char c;
   };
typedef struct node NODE /* NODE może być używane zamiast node */
```

Można by pomyśleć, że poniższa skrócona deklaracja będzie lepsza, ale jest ona błędna!

```
typedef struct node      /* TA DEKLARACJA JEST BŁĘDNA!!! */
   {
      NODE *parent;
      NODE *right_child;
      NODE *left_child;
      char c;
   } NODE;
```

Ten jednostopniowy proces jest błędny, ponieważ kompilator spotyka pierwsze użycie nazwy NODE zanim zostało zakończone jej definiowanie.

Przyjęto pisać nazwy nowych typów wielkimi literami, tak jak w powyższych przykładach, dzięki czemu wyróżniają się one w programie.

PATRZ TAKŻE
struktura

---

unie

Unia jest to zmienna, która służy do pamiętania kilku obiektów różnych typów w tym samym obszarze pamięci. Unii używa się tam gdzie jest ważna zajętość pamięci albo tam, gdzie sposób operowania danymi jest taki, że obiekty o różnych typach powinny zajmować ten sam obszar pamięci. Unie nie są zależne od implementacji, to znaczy można je używać w różnych komputerach, mimo że szczegóły ich realizacji mogą być różne dla różnych komputerów. Unię można zadeklarować na jeden z trzech sposobów.

1. union *oznacznik* (*lista_deklaracji*);

Deklaracja unii o nazwie *oznacznik* bez rezerwowania dla niej pamięci i bez definiowania żadnej zmiennej.

2. union {*lista_deklaracji*} *nazwa_zmiennej*;

Deklaracja nie nazwanej unii i zdefiniowanie pojedynczej zmiennej (lub tablicy zmiennych), których typ jest taki jak typ unii.

3. union *oznacznik {lista_deklaracji} nazwa_zmiennej*;

Kombinacja dwóch powyższych przypadków: deklaracja unii i zdefiniowanie zmiennej o takim typie.

W unii przewiduje się dostatecznie dużo miejsca, aby mógł w niej zmieścić się każdy z jej elementów. Można na unię patrzeć jak na swoistą strukturę, w której nie występuje przesunięcie elementów od początku; w strukturze elementy znajdują się jedne za drugimi.

Programista musi zawsze wiedzieć, jaki element jest umieszczony w unii w danym momencie; można zdefiniować dodatkową zmienną, której wartość określa typ zmiennej ostatnio zapamiętanej w unii.

PRZYKŁADY

```
union u_nazwa
{
    int i;
    char c;
};
```

Zadeklarowano tu unię o nazwie u_nazwa, ale nie zdefiniowano żadnej zmiennej typu union u_nazwa. Definicje podane poniżej są poprawne pod warunkiem, że odbywają się w zasięgu powyższej deklaracji:

```
union u_nazwa c1;
union u_nazwa *c1;
union u_nazwa *func();
union u_nazwa c[2];
```

W pierwszej definiuje się zmienną typu union u_nazwa, w drugiej – wskaźnik do typu union u_nazwa; w trzeciej deklaruje się funkcję dającą w wyniku typ union u_nazwa, a w ostatniej definiuje się tablicę dwóch zmiennych typu union u_nazwa.

```
union
    {
        int i;
        char c;
    } byte;
```

Zadeklarowano tu nie nazwaną unię i zdefiniowano pojedynczą zmienną tego typu. Ponieważ ta unia nie ma oznacznika, nie jest więc możliwe zdefiniowane innych zmiennych lub wskaźników do struktury tego typu.

```
union u_nazwa
    {
        int i;
        char c;
    } byte[2];
```

Zadeklarowano tu unię o nazwie u_nazwa i zdefiniowano tablicę dwóch zmiennych typu union u_nazwa. Definicje i deklaracje podane dla pierwszego przykładu są także

tutaj dopuszczalne, pod warunkiem jednak, że znajdują się w zasięgu powyższej deklaracji.

Elementami unii mogą być zmienne dowolnego typu, łącznie ze strukturami i innymi uniami. Na przykład:

```
struct
   {
      int i;
      int j;
      union
        {
          int q;
          char *s;
            struct
              {
                int m;
                char *t;
              } su_przyklad;
        } u_przyklad;
      char *p;
   } s_przyklad [MAX_STRUKTURA];
```

Do elementu m odwołujemy się jak następuje:

```
s_przyklad[j].u_przyklad.su_przyklad.m
```

Do poszczególnych elementów unii można odwoływać się w różny sposób. Przy podanych poniżej deklaracjach i definicjach:

```
union u_nazwa
   {
      int i;
      char c;
   } byte;
union u_nazwa *p_u = &byte;
```

wyrażenia byte.i, (*p_u).i oraz p_u->i dotyczą tej samej zmiennej i w unii byte. Wyrażenie byte.i znaczy "zmienna i w unii union byte".

Wyrażenie (*p_u).i znaczy "zmienna i w unii wskazywanej przez p_u". Nawiasy są niezbędne ponieważ priorytet operatora . jest wyższy niż priorytet operatora *.

Wyrażenie p_u->i ma dokładnie to samo znaczenie co (*p_u).i, a ponieważ w tej postaci stosuje się tylko jeden operator, unika się więc kłopotów z priorytetami operatorów i wobec tego należy tę postać preferować. Operator -> składa się ze znaku łącznika -, po którym bez odstępu następuje znak większości > .

Ponieważ operatory . i -> znajdują się w grupie pierwszej operatorów o najwyższym priorytecie, trzeba więc uważać przy używaniu ich w połączeniu z innymi operatorami w wyrażeniach. Na przykład ++p_u->i spowoduje inkrementację

zmiennej i, a nie p_u. W celu wykonania inkrementacji p_u przed pobraniem zmiennej i należy napisać (+ +p_u)->i.

Jedyne operacje jakie można wykonywać na unii to znalezienie jej adresu (za pomocą operatora &) i odwołanie się do jej elementów (za pomocą operatora . lub kombinacji operatorów * i . albo ich odpowiednika ->). W niektórych kompilatorach jest możliwe przekazanie unii jako argumentu funkcji.

Przy odwołaniu do elementów unii zakłada się, że wyrażenie po lewej stronie operatora . lub -> wskazuje unię, której elementem jest argument po prawej stronie operatora. W rzeczywistości to całkiem rozsądne wymaganie wcale nie jest wymuszane. Wyrażenie po lewej stronie operatora . może być dowolną l-wartością całkowitą a wyrażenie po lewej stronie operatora -> może być wskaźnikiem do dowolnej liczby całkowitej. Innymi słowy, z punktu widzenia kompilatora cokolwiek może wskazywać na cokolwiek. Kompilator w żaden sposób nie sprawdza w czasie kompilacji, czy programista używa właściwego wskaźnika; z całą pewnością też nie sprawdza się w czasie działania programu, czy wskaźnik ma "sensowną wartość". Zadaniem programisty jest zapewnienie, aby wskaźniki wskazywały właściwe obiekty.

PATRZ TAKŻE

pola bitowe

struktura

Unix

Unix jest systemem operacyjnym ściśle związanym z językiem C. W samej rzeczy, Unix i język C zostały opracowane przez tych samych ludzi w Bell Laboratories (por. rozdz. 2). Oto główne cechy systemu operacyjnego Unix.

1. System plików jest zorganizowany w hierarchiczną strukturę drzewiastą o dostępie niezależnym od urządzenia. Oddzielne zezwolenia na odczyt, zapis i wykonanie zapewniają łatwy dostęp i sensowną ochronę.

2. Narzędzia mają postać wielu małych programów, a nie kilku dużych. Powody takiej realizacji są historyczne, ale jej zaletą jest fakt, że łatwo jest połączyć kilka małych programów w celu wykonania dużej i złożonej pracy. Co więcej, wiele z funkcji realizowanych przez te programy jest także dostępnych jako funkcje systemowe, z których można korzystać w programach napisanych w języku C. Wadą takiego podejścia jest fakt, że wiele z tych małych programów było pisanych przez różnych ludzi i używa się w nich różnych, niezgodnych i utrudniających użycie sposobów konwersacji z użytkownikiem.

3. Potoki (ang. *pipes*) umożliwiają przesłanie wyjścia z jednego programu bezpośrednio na wejście następnego programu. Ta właściwość systemu jest kompletnie niewidoczna dla tych programów i programista nie musi robić niczego specjalnego aby z niej korzystać.

4. Pliki "standardowe" (standardowe wejście, standardowe wyjście i standardowe wyjście błędów) są automatycznie definiowane dla każdego programu; podczas wykonywania programu można je skierować do innych urządzeń. Takie skierowanie, podobnie jak potoki, jest całkowicie niewidoczne dla tych programów.

5. Około 95% kodu systemu Unix zostało napisane w języku C, dzięki czemu można go łatwo przenosić, o ile w ogóle można mówić o przenośności systemu operacyjnego.

6. Pewien brak jednorodności i symetrii jest odbiciem braku dostatecznego planowania w fazie jego opracowania. Dokumentacja jest często post factum opisem z obserwacji zachowania się programu niż wzorcem dla jego przewidywanego zachowania.

7. Zwięzły "nieprzyjazny użytkownikowi" dialog z użytkownikiem utrudnia początkującym nauczenie się systemu. Z drugiej strony, napisanie dobrego i łatwego w użyciu interpretatora poleceń (znanego pod nazwą "shell") jest dość proste.

Reasumując, system operacyjny Unix tworzy niesłychanie wydajne środowisko do opracowywania programów, o wciąż rosnącej popularności. Unix jest dostępny dla coraz większej liczby typów komputerów, czasami jako drugi system operacyjny, pracujący pod kontrolą podstawowego systemu, dostarczanego przez producenta sprzętu. Ostatnio pojawiło się wiele systemów "wyglądających jak Unix".

Opracowanie programu w języku C pod kontrolą systemu operacyjnego Unix umożliwia dość łatwe przeniesienie go na inne komputery. Nigdy nie odbywa się to bezboleśnie i całkiem prosto, ale jest na ogół znacznie łatwiejsze niż w przypadku innych systemów operacyjnych i innych języków programowania.

---

ustawienie

Zagadnienie to dotyczy jedynie agregatów.

Sprzęt niektórych komputerów wymaga, aby zmienne pewnego typu były ustawiane (zaczynały się) na granicy słowa lub podwójnego słowa (tj. miały adresy podzielne odpowiednio przez 2 lub przez 4). Kompilator na ogół spełnia te wymagania, nawet jeśli oznacza to zostawianie "dziur" między składnikami agregatu.

Regułą jest, że program nie może przyjmować żadnych założeń dotyczących kolejności zapamiętywane są składników w agregacie, łącznie z polami bitowymi.

PATRZ TAKŻE

pola bitowe
agregat

---

void

Ten typ danych może być stosowany tylko do oznaczania typu wyniku przekazywanego przez funkcję. Jeśli funkcja została tak zdefiniowana, to nie może ona zawierać instrukcji return *wyrazenie*, nie można też nigdzie w programie korzystać z przekazywanego przez tę funkcję wyniku.

Typ void jest dostępny nie we wszystkich implementacjach C.

PRZYKŁAD

```
void func(i)
int i;
```

```
    {
        return i;    /* TO JEST BŁĄD!!! */
        return;      /* to jest poprawne */
    }
if (j = func(i))     /* TO TEŻ JEST BŁĄD!!! */
```

---

while

Składnia instrukcji while jest następująca:

while (*wyrażenie*) *instrukcja*

Gdy *wyrażenie* nie jest zerowe, tzn. gdy jego obliczenie daje wartość różną od zera, to jest wykonywana instrukcja.

Instrukcja while jest zbliżona do instrukcji do .. while, z wyjątkiem tego, że jeśli przy pierwszym obliczaniu *wyrażenie* ma wartość zerową, to w pętli while *instrukcja* nie jest wykonywana ani razu, a w pętli do .. while jest wykonywana co najmniej raz; wynika to z faku, że w pętli do .. while *wyrażenie* jest obliczane dopiero po pierwszym wykonaniu *instrukcji*.

Instrukcja może być instrukcją złożoną.

PRZYKŁAD

```
while (i < k)
    {
        func_jeden(i++);
        func_dwa(j++);
    }
```

PATRZ TAKŻE

do .. while
break
continue

---

wiersz kontynuacji

Znak ukośnika \ umożliwia kontynuowanie instrukcji w następnym wierszu.

Kompilator nie wymaga znaku ukośnika, z wyjątkiem przypadku ciągu znaków, a preprocesor wymaga go w przypadku makrodefinicji.

PRZYKŁAD

```
char *str = "abc\
    def";
#define DUZE_MAKRO(a,b,c) { func_a((a),(b),(c)); \
                            func_b((a)); }
```

---

wskaźnik do funkcji

Kompilator przyjmuje, że nazwa funkcji, jeśli pojawia się jako pierwszy element wyrażenia, jest wywołaniem funkcji o tej nazwie. W pozostałych przypadkach przyjmuje się, że nazwa funkcji jest wskaźnikiem do funkcji.

Przykładem użycia wskaźnika do funkcji jest przekazanie argumentu do innej funkcji. Na przykład:

```
func1();          /* deklaracja funkcji */
int q;
func2(func1, &q); /* wywołanie func2 z argumentem w postaci wskaźnika
                     do func1 */

func2(f, k)
int (*f)();
int *k;
{
    int i;
    i = *k
    *(f)(i);
}
```

Funkcja func1 musi być zadeklarowana, nawet jeśli w wyniku daje liczbę całkowitą, gdyż bez deklaracji kompilator przyjąłby, że jest to nazwa zmiennej w instrukcji func2(func1, &q);.

Zauważmy, że deklaracja int (*f)() w funkcji func2 oznacza, że jest to deklaracja wskaźnika do funkcji dającej w wyniku liczbę całkowitą. Pominięcie pierwszej pary nawiasów, to jest deklaracja postaci int *f() oznacza funkcję dającą w wyniku wskaźnik do liczby całkowitej.

Przyjrzyjmy się sposobowi użycia f w funkcji func2. Do tej funkcji odwołujemy się przez *f; w podobny sposób odwołalibyśmy się do zmiennej typu int w funkcji, do której ta liczba całkowita została przekazana przez wskaźnik.

Oto inny przykład użycia wskaźnika do funkcji:

```
int (*function[NUM_FUNCTIONS])();  /* definicja tablicy */
int func_0();           /* deklaracja funkcji */
int func_1();           /* deklaracja funkcji */
    .
    .
    .
for (i = 0; i<NUM_FUNCTIONS; i++) /* nadaje wartość początkową */
    function[i] = NULL;
function[0] = func_0;       /* ładuje do tablicy wskaźniki do funkcji */
function[1] = func_1;
    .
    .
(*function[j])(i,j,k);      /* wywołanie funkcji */
```

Podana wyżej metoda jest użyteczna w przypadku użycia jednej spośród dużej liczby funkcji z tymi samymi argumentami.

---

wskaźnik do zmiennej

Wskaźnik do zmiennej jest to zmienna przyjmująca jako swoją wartość adres innej zmiennej. Wskaźniki są używane w następujących sytuacjach.

1. Przetwarzanie tablic i ciągów znaków (tablic znaków).

Tablice są ściśle związane ze wskaźnikami i mogą być używane zamiennie. Na przykład definicja

```
char c[8]
```

definiuje tablicę 8 zmiennych typu char, o nazwach od c[0] do c[7]. Definicja

```
char *string
```

definiuje wskaźnik o nazwie string do zmiennej typu char. Wyrażenie

```
*string
```

oznacza "znak wskazywany przez string".

W rzeczywistości *wyrażenie1*[*wyrażenie2*] jest rozumiane jako *((*wyrażenie1*) + (*wyrażenie2*)).

PRZYKŁAD

Przyjmijmy następujące definicje:

```
char c[8] = {'a','b','c','d','e','f','g','\0'};
char *string = c;
```

Pierwsza definicja określa tablicę znakową typu char o nazwie c, złożoną z 8 elementów, której przypisano wartości początkowe od 'a' do 'g' i zakończono znakiem końca ciągu znaków. Druga definicja określa zmienną string, wskaźnik do zmiennej typu char i nadaje jej wartość początkową c, tak aby wskazywała tablicę c. (Zauważmy, że istnieje różnica między c – adresem pierwszego elementu tablicy c, a c[0] – pierwszym elementem tablicy c.)

Istnieją dwie metody odwołania się do czwartego elementu tablicy c:

```
c[3]
```

lub

```
*(string + 3)
```

Pierwsza metoda jest typowa dla odwołań do tablic. Drugie odwołanie – *(string + 3) – należy czytać "znak wskazywany przez (string + 3)"; odwołuje się ono także do elementu c[3], gdyż string wskazuje c[0], a trzy znaki dalej znajduje się znak c[3].

Przyjmijmy poniższe definicje:

```
int i[8] = {0,1,2,3,4,5,6,7};
int *p_i = i;
```

Pierwsza definicja określa tablicę i liczb całkowitych o 8 elementach i nadaje im wartości początkowe od 0 do 7. Druga definicja określa p_i jako wskaźnik do zmiennej

typu int i nadaje mu taką wartość początkową, aby wskazywał tablicę i. (Zauważmy, że istnieje różnica między i – adresem pierwszego elementu tablicy i, a i[0] – pierwszym elementem tablicy i.)

Istnieją dwie metody odwołania się do czwartego elementu tablicy i:

```
i[3]
```

lub

```
*(p_i + 3)
```

Pierwsza metoda jest typowa dla odwołań do tablic. Drugie odwołanie – *(p_i + 3) – należy czytać "liczba całkowita wskazywana przez (p_i + 3)"; odwołuje się ono także do elementu i[3], gdyż p_i wskazuje i[0], a trzy liczby całkowite dalej znajduje się liczba i[3].

Zauważmy, że p_i + 1 wskazuje nie komórkę pamięci o jeden bajt dalej niż komórka wskazywana przez p_i, ale komórkę położoną dalej o rozmiar zmiennej typu int.

Wartości początkowe do tablicy można wpisać za pomocą następującej instrukcji for:

```
for (j = 0; j < 8; j++)
*(p_i + j) = 0;
```

2. Sytuacje, w których wymagania na rozmiar pamięci nie da się określić w chwili kompilacji, ale pamięć musi być przydzielana dynamicznie. Nie zawsze wiadomo dokładnie, jak duża tablica jest potrzebna. Ponadto, czasem trzeba mieć do dyspozycji dużą liczbę tablic w różnych miejscach programu i o różnym czasie życia; powtórne użycie pamięci jest więc zalecane, zwłaszcza w komputerach o małej pojemności pamięci.

Jedno z rozwiązań polega na dynamicznym łączeniu i przydzielaniu pamięci strukturom, jak w poniższym przykładzie:

```
struct link
    {
        /* zmienne danych */
    struct link *next;  /* wskaźnik do następnej struktury w liście */
    struct link *prev;  /* wskaźnik do poprzedniej struktury w liście */
    }
struct link *first;  /* wskaźnik do pierwszej struktury w liście */
struct link *last;   /* wskaźnik do ostatniej struktury w liście */
```

Dla każdej struktury link można przydzielić pamięć wywołując jedną z funkcji przydziału pamięci, na przykład funkcję malloc. Programista musi jedynie umożliwić poprawne łączenie struktur w łańcuch za pomocą wskaźników. Jeśli link nie jest potrzebny, to można go odłączyć od łańcucha i oddać pamięć do systemu operacyjnego za pomocą funkcji free. Ten sposób umożliwia operowanie małą pamięcią.

3. Umożliwienie funkcjom modyfikowania zmiennych, które są dla nich niewidoczne.

W języku C funkcja nie ma dostępu do swoich aktualnych argumentów, gdyż wywołania są realizowane przez wartość. Z drugiej strony funkcja może zmodyfikować zmienną, której nie widzi, pod warunkiem, że widzi wskaźnik do niej, jeśli ten wskaźnik został zdefiniowany globalnie lub jest jednym z argumentów funkcji.

PRZYKŁAD

```
int i;
int k;
func(&i, k);
func(j, n)
int *j;
int n;
{
    *j = 1        /* nadanie i wartości 1 */
    n = 0;        /* nie ma wpływu na k */
}
```

Zauważmy, że gdy wskaźnik jest argumentem, to funkcja nie może zmodyfikować samego wskaźnika, ale tylko wskazywany przez niego obiekt. Jeśli funkcja ma zmodyfikować wskaźnik, to musi ona widzieć wskaźnik do tego wskaźnika.

---

wyrażenie

Wyrażenie może mieć jedną z następujących postaci:
- nazwa zmiennej
- nazwa tablicy; jeśli na przykład s zostało zdefiniowane jako char s[10], to s jest wyrażeniem rozumianym jako adres s[0];
- stała, na przykład 10;
- wywołanie funkcji, na przykład sqrt(d);
- odwołanie do elementu struktury, na przykład x.a lub p–>a;
- odwołanie do elementu tablicy, na przykład s[2];
- nazwa funkcji, gdy jest przekazywana w postaci wskaźnika w wywołaniu innej funkcji;
- jedno z powyższych wyrażeń z zastosowaniem operatora i (lub) nawiasów, na przykład (i), i++, s[2] – 'a', i == 0 lub i + 1.

---

wywołanie przez nazwę i przez wartość

W języku C argumenty są przekazywane przez wartość, nie przez nazwę. Wywołana funkcja otrzymuje kopię argumentu i nie jest w stanie zmienić wartości oryginalnej zmiennej.

Funkcja może jednak zmienić wartość niewidocznej dla niej zmiennej, jeśli widzi ona wskaźnik do tej zmiennej, tj. jeśli ten wskaźnik jest zmienną globalną lub został przekazany funkcji jako jej argument. Zauważmy, że w tym drugim przypadku,

sam wskaźnik jest przekazywany przez wartość, a wywołana funkcja nie może zmienić wartości wskaźnika.

Nazwa tablicy jest rozumiana jako jej adres; jeśli więc nazwa tablicy jest przekazywana jako argument funkcji, to faktycznie jest przekazywany adres tej tablicy. W tym przypadku wywoływana funkcja może zmienić wartości w tablicy.

W tych wersjach C, w których jest możliwe przekazywanie struktury jako argumentu wywołania funkcji, faktycznie jest przekazywany jej adres.

PRZYKŁAD

```
main()
{
    register i;
    char c[10];    /* definiuje tablicę 10-elementową */
    int j;
    int k;
    int *p_k = &k;    /* definiuje wskaźnik do liczby całkowitej, ustawia
                         jego wartość początkową p_k tak, aby k i* p_k
                         odpowiadały temu samemu obiektowi */
func(&j,&p_k);        /* wywołanie funkcji z dwoma argumentami:
                         1) adres j
                         2) adres p_k (adres adresu k) */
}
func(q,r)
int *q;               /* wskaźnik do liczby całkowitej */
int **r;              /* wskaźnik do wskaźnika do liczby całkowitej */
{
    *q = 0;
    q = 0;
    **r = 0;
    *r = 0;
    r = 0;
}
```

Wyrażenie &j oznacza "adres j".

Funkcja func została wywołana z argumentami j oraz p_k. Zwróćmy uwagę na deklarację formalnego argumentu q jako wskaźnika do liczby całkowitej oraz r jako wskaźnika do wskaźnika do liczby całkowitej. Instrukcja:

```
*q = 0;
```

nadaje wartość zero liczbie całkowitej (j w funkcji main), której adres jest podany przez q. Instrukcja

```
q = 0;
```

nadaje q wartość równą zeru, co oznacza, że ten wskaźnik nie wskazuje już żadnego obiektu, ani nie ma wpływu na wartość zmiennej w funkcji main.

Zmienna r wskazuje p_k w funkcji main, samo zaś p_k wskazuje k. Instrukcja

```
**r = 0;
```

nadaje wartość zero liczbie całkowitej (k w funkcji main), której adres (podawany przez p_k w funkcji main) jest wskazywany przez r. Instrukcja

```
*r = 0;
```

nadaje wartość zero wskaźnikowi wskazywanemu przez r, to jest p_k w funkcji main. Instrukcja

```
r=0;
```

nadaje r wartość zero, wskutek czego nie wskazuje on już żadnego obiektu, ani nie ma wpływu na wartość zmiennej w funkcji main.

PRZYKŁAD

```
func_jeden()
{
    int a[10];
    func_dwa(a);
}
func_dwa(a)
int a[];
{
}
```

Do funkcji func_dwa przekazuje się jako argument adres pierwszego elementu tablicy a, innymi słowy stałą. Korzystając z tego adresu funkcja func_dwa może bezpośrednio sięgać do dowolnego elementu w tej tablicy.

Zauważmy, że rozmiar tablicy jest określany w deklaracji argumentów funkcji func_jeden, a nie w samej funkcji func_dwa.

---

zasięg

Zasięg zmiennej jest to ta część pliku źródłowego, w której nazwa zmiennej jest widoczna dla kompilatora (lub jest mu znana).

Nazwa zmiennej staje się znana kompilatorowi od tego miejsca w pliku źródłowym, w którym zostaje zdefiniowana lub zadeklarowana, i przestaje dla niego istnieć w końcu bloku lub funkcji, w której została zdefiniowana lub zadeklarowana, lub w końcu pliku źródłowego, w którym została zdefiniowana lub zadeklarowana – zależnie od tego, co następuje wcześniej.

Zasięg deklaracji funkcji (w odróżnieniu od definicji funkcji) jest określony tymi samymi regułami, co zasięg zmiennej.

PRZYKŁAD

```
/* początek pliku źródłowego */
int i;
int n;
extern int q;
```

```
main()
{
    int j;
    int n;
    char *char_func();
    .
    .
    .
}
int k;
func()
{
}
/* koniec pliku źródłowego */
```

Zmienna i jest zdefiniowana globalnie dla całego pliku źródłowego; jej zasięg rozciąga się na cały plik źródłowy.

Zmienna q jest zadeklarowana globalnie i jej zasięg rozciąga się na cały plik źródłowy. Zmienna q musi być zdefniowana w innym pliku źródłowym, który będzie dołączony przez konsolidator; w przeciwnym razie konsolidator wskaże błąd, gdyż nie znajdzie jej definicji.

Zmienna j jest zdefiniowana lokalnie w funkcji main i jej zasięg jest ograniczony do tej funkcji. Zasięg zmiennej k zaczyna się od punktu, w którym została zdefiniowana i kończy z końcem pliku źródłowego.

Zauważmy, że istnieją dwie zmienne n, jedna zdefiniowana globalnie, a druga lokalna dla main. Wewnątrz funkcji main nie ma możliwości odwołania się do zdefiniowanej globalnie zmiennej n. Każde odwołanie do n wewnątrz main dotyczy zmiennej lokalnej, a każde odwołanie do n poza funkcją main dotyczy zmiennej globalnej. To ograniczenie pokazuje, że należy unikać używania powtarzających się nazw zmiennych.

W celu uzyskania dostępu do tej samej zmiennej i w innym pliku źródłowym, należy zadeklarować ją jako extern w tym drugim pliku. Jeśli nie została zadeklarowana jako extern, ale zdefiniowana (jako int i), to mogą istnieć dwie zmienne o nazwie i, oddzielnie w dwóch plikach źródłowych.

W zasadzie nie ma możliwości bezpośredniego odwołania się do zmiennej j spoza funkcji main. Można się do niej odwołać, jeśli przekaże się ją jako argument do innej funkcji, a także można zmienić jej wartość, jeśli jej adres zostanie przekazany jako argument do innej funkcji, ale nie można się do niej odwoływać bezpośrednio spoza funkcji main.

PATRZ TAKŻE

czas życia zmiennej
deklaracja extern
argument formalny i aktualny

---

7. Dyrektywy preprocesora

#define

Składnia instrukcji #define jest następująca:

#define *identyfikator(identyfikator, ...)*

lub

#define *identyfikator znacznik*

W obydwu postaciach słowo kluczowe #define musi znajdować się na samym początku wiersza. Pierwsza postać tworzy makrodefinicję; jej użycie omówiono w haśle makrodefinicja. Druga postać służy do definiowania stałych symbolicznych; na przykład instrukcja

#define MAXLINES 128

zamienia wszystkie wystąpienia ciągu znaków MAXLINES na ciąg 128, aż do końca pliku źrodłowego lub aż do wystąpienia dyrektywy #undef. Zauważmy, że *znacznik* jest opcjonalny, tak że dopuszczalna jest też definicja:

#define MAXLINES

Długą definicję można kontynuować w następnym wierszu, jeśli ten poprzedni zakończy się znakiem ukośnika \.

W definicji można korzystać z poprzednich definicji. Także *znacznik* jest sprawdzany wielokrotnie w celu zamiany wszystkich zdefiniowanych identyfikatorów. Na przykład definicje:

#define AAA BBB
#define BBB CCC

spowodują, że preprocesor zamieni wszędzie AAA na CCC.

Podstawianie nie odbywa się wewnątrz ciągów znaków zawartych w cudzysłowach.

Zdefiniowany identyfikator może być przedefiniowany kolejną dyrektywą #define. Jeśli *znacznik* jest różny w tych dwóch definicjach, to preprocesor ostrzega o tym fakcie.

Dyrektywy #define można użyć do dokonywania dowolnie złożonych podstawień. Korzystając z tej dyrektywy programista może osiągnąć zdumiewające efekty, ale płaci za to trudnościami w uruchamianiu, gdyż plik, który ogląda za pomocą edytora tekstowego, jest różny od pliku przesyłanego do kompilatora.

PATRZ TAKŻE

#undef

#else patrz *#if*

#endif patrz *#if*

#if

Istnieje kilka postaci tej dyrektywy preprocesora.

1. #if *ograniczone_wyrażenie_całkowitoliczbowe* – określa czy to *ograniczone_wyrazenie_całkowitoliczbowe* daje w wyniku wartość niezerową. *Ograniczone_wyrażenie_całkowitoliczbowe* jest to takie wyrażenie, które może być obliczone przez preprocesor, na przykład 1 + 2.

W powyższym wyrażeniu nie mogą występować wywołania funkcji, konwersja typów, operator sizeof i stałe wyliczeniowe, gdyż preprocesor nie jest w stanie takiego wyrażenia obliczyć (te wyrażenia mogą być wyliczone tylko podczas działania programu).

2. #ifdefined *identyfikator*

3. #ifdefined(*identyfikator*)

4. #ifdef identyfikator

Te trzy postaci są równoważne i dają w wyniku wartość niezerową, jeśli *identyfikator* był poprzednio zdefiniowany (z użyciem dyrektywy #define) w pliku źródłowym.

5. #ifndef *identyfikator*

Powyższe wyrażenie daje wynik niezerowy, jeśli *identyfikator* nie był poprzednio zdefiniowany w pliku źródłowym.

We wszystkich tych dyrektywach znak # musi być pierwszym znakiem w wierszu.

Jeżeli w wyniku obliczenia instrukcji #if otrzymuje się wartość niezerową, to następne wiersze programu źródłowego – aż do dyrektywy #endif lub #else – są włączane do pliku podlegającego kompilacji. Jeżeli w wyniku obliczenia instrukcji #if otrzymuje się wartość zerową, to następne wiersze programu źródłowego – aż do dyrektywy #endif lub #else – nie są włączane do pliku podlegającego kompilacji.

PRZYKŁADY

```
#ifdef DEBUG
print_table();
#endif
```

Jeśli wcześniej została zdefiniowana stała DEBUG, to wiersz zawarty między #ifdef a #endif podlega kompilacji.

```
#ifndef DVP
.
.
#else
.
.
#endif
```

Jeśli wcześniej została zdefiniowana stała DVP, to wiersze zawarte między #else a #endif podlegają kompilacji, w przeciwnym razie kompilacji podlegają wiersze zawarte między #ifndef a #else.

#ifdef patrz *#if*

#include

Dyrektywa #include powoduje, że kompilator czyta pliki o podanych nazwach jakby były częścią pliku kompilowanego. Składnia tej dyrektywy jest następująca:

#include "*nazwa pliku*"

lub

#include <*nazwa pliku*>

Pierwszą postać stosuje się wówczas, gdy plik o podanej nazwie znajduje się w bieżącym katalogu. Drugą postać stosuje się wówczas, gdy plik o podanej nazwie ma być odczytany z katalogu uważanego za standardowy (w systemie operacyjnym Unix jest to katalog /usr/include).

Dyrektywy #include mogą być dowolnie zagnieżdżone, to jest włączany plik może także zawierać dyrektywy #include itd.; ograniczeniem jest jedynie dopuszczalna liczba plików, jakie system operacyjny może otworzyć naraz.

PRZYKŁADY

```
#include "abc.h"
#include "../DVP/TRAN/abc.h"
#include <ctime.h>
#include <sys/stat.h>
```

#line

Składnia tej dyrektywy jest następująca:

#line *stała nazwa_pliku*

Wykonanie tej dyrektywy powoduje, że kompilator przechodzi do wiersza o numerze określonym przez *stałą* w pliku o podanej *nazwie*. Jest to dyrektywa użyteczna

w programach generujących programy w języku C, gdyż umożliwia prześledzenie źródła powstawania błędów.

#undef

Dyrektywa #undef powoduje, że identyfikator określony wcześniej za pomocą dyrektywy #define nie jest w dalszej części tekstu uważany za zdefiniowany. Składnia tej dyrektywy jest następująca:

#undef *identyfikator*

Dodatek A. Funkcje standardowe

We wszystkich implementacjach C jest dostępna większość z opisanych tu funkcji, chociaż mogą się one różnić w szczegółach. Programista powinien przed użyciem tych funkcji dokładnie przeczytać opis używanej implementacji. Funkcje opisane w dodatku są podzielone na następujące grupy.

1. Funkcje wejścia-wyjścia

– funkcje wejścia-wyjścia niebuforowane (niskiego poziomu)

close	zamyka plik
creat	tworzy plik
eof	sprawdza znacznik końca pliku
lseek	zmienia bieżącą pozycję w pliku
open	otwiera plik
read	czyta z pliku
tell	podaje wartość bieżącej pozycji w pliku
write	zapisuje do pliku

– funkcje wejścia-wyjścia buforowane

clearerr	zeruje znacznik błędu
fclose	zamyka plik
feof	daje w wyniku TRUE, gdy koniec pliku
ferror	daje w wyniku TRUE, gdy wystąpił błąd operacji na pliku
fflush	wymusza zapisanie bufora
fgetc	czyta kolejny znak z pliku
fgets	czyta znaki z pliku
fileno	podaje deskryptor pliku
fopen	otwiera plik
fprintf	zapisuje argumenty do pliku
fputc	zapisuje znak do pliku
fputs	zapisuje ciąg znaków do pliku
fread	czyta bloki danych z pliku do tablicy znaków

freopen	zamyka plik i otwiera inny
fscanf	czyta argumenty z pliku
fseek	zmienia bieżącą pozycję pliku
ftell	podaje przesunięcie od początku pliku
fwrite	zapisuje bloki danych do pliku
getc	czyta kolejny znak z pliku
getchar	czyta kolejny znak z pliku stdin
gets	czyta znaki z pliku stdin do zmiennej znakowej
getw	czyta słowo (liczbę całkowitą) z pliku
printf	zapisuje argumenty do pliku
putc	zapisuje znak do pliku
putchar	zapisuje znak do pliku stdout
puts	zapisuje ciąg znaków do pliku stdout
putw	zapisuje słowo (liczbę całkowitą) do pliku
scanf	czyta argument z pliku stdin
sscanf	czyta argument z ciągu znaków
tmpfile	tworzy plik tymczasowy i otwiera go do zapisywania
ungetc	powoduje wycofanie znaku do pliku, tak że następne wykonanie getc da ponownie ten znak
unlink	usuwa plik

2. Funkcje przetwarzania ciągów znaków

index	znajduje pierwsze wystąpienie znaku w ciągu
memchr	znajduje pierwsze wystąpienie znaku w początkowych n znakach ciągu
memcmp	porównuje początkowe n znaków ciągów
memccpy	kopiuje kolejne znaki z jednego ciągu do drugiego, aż do skopiowania zadanego znaku albo zadanej liczby znaków
memcpy	kopiuje n znaków z jednego ciągu do drugiego
memset	ustawia n znaków na zadaną wartość
rindex	znajduje ostatnie wystąpienie znaku w ciągu
sprintf	"zapisuje" argumenty do ciągu
strcat	łączy dwa ciągi
strchr	znajduje pierwsze wystąpienie znaku w ciągu
strcmp	porównuje dwa ciągi
strcpy	kopiuje jeden ciąg do drugiego
strlen	podaje długość ciągu
strncat	dołącza znaki jednego ciągu do drugiego
strncmp	porównuje znaki w ciągach
strncpy	kopiuje znaki z jednego ciągu do drugiego
strrchr	znajduje ostatnie wystąpienie znaku w ciągu

3. Funkcje przetwarzania znaków

isalnum	sprawdza czy znak jest alfanumeryczny
isalpha	sprawdza czy znak jest literą
isascii	sprawdza czy znak należy do zbioru kodów ASCII

iscntrl	sprawdza czy znak jest znakiem sterującym
isdigit	sprawdza czy znak jest cyfrą
isgraph	sprawdza czy znak jest znakiem graficznym
islower	sprawdza czy znak jest małą literą
isprint	sprawdza czy znak daje się wydrukować
ispunct	sprawdza czy znak jest znakiem interpunkcyjnym
isspace	sprawdza czy znak jest zdefiniowany jako odstęp
isupper	sprawdza czy znak jest wielką literą
toascii	przekształca znak na znak kodu ASCII
tolower	zamienia wielką literę na małą
toupper	zamienia małą literę na wielką

4. Funkcje matematyczne

abs	wartość bezwzględna
acos	arcus cosinus
asin	arcus sinus
atan	arcus tangens
atan2	arcus tangens (wariant)
atoi	przekształca ciąg znaków na liczbę całkowitą
atof	przekształca ciąg znaków na liczbę rzeczywistą
atol	przekształca ciąg znaków na liczbę całkowitą długą
ceil	podaje najmniejszą liczbę całkowitą nie mniejszą niż d
cos	cosinus
cosh	cosinus hiperboliczny
exp	e do potęgi d
fabs	wartość bezwzględna d
floor	największa liczba całkowita nie większa niż d
fmod	dzielenie modulo (reszta z dzielenia)
log	logarytm naturalny
log10	logarytm dziesiętny
pow	potęga
rand	liczba pseudolosowa
random	liczba pseudolosowa
sin	sinus
sinh	sinus hiperboliczny
sqrt	pierwiastek kwadratowy
srand	ustawia generator liczb pseudolosowych
tan	tangens
tanh	tangens hiperboliczny

5. Funkcje przydzielania i zwalniania pamięci

calloc	przydziela pamięć
free	zwalnia pamięć
malloc	przydziela pamięć
realloc	zmienia przydział pamięci

6. Funkcje daty i czasu

clock	mierzy czas
ctime	przekształca czas na ciąg znaków
gmtime	przekształca czas na czas GMT (średni czas Greenwich)
localtime	przekształca czas na czas lokalny
time	podaje datę i czas

7. Funcje różne

system	wykonuje polecenie systemu operacyjnego zawarte w ciągu znaków
exit	kończy wykonanie programu po zapisaniu buforów
_exit	kończy wykonanie bez zapisywania buforów
qsort	sortuje

8. Format tekstu sterującego funkcjami printf
9. Format tekstu sterującego funkcjami scanf

Funkcje są opisane w kolejności alfabetycznej wewnątrz każdej grupy w następujący sposób.

<u>Opis</u>	<u>Znaczenie</u>
fclose zamyka plik	nazwa funkcji i jej znaczenie
int fclose(wsk_pliku)	typ funkcji, nazwa funkcji, argumenty
FILE *wsk_pliku;	deklaracje argumentów
Dołączyć plik:	pliki .h, które powinny być włączone
Daje w wyniku:	wartość przekazywana przez funkcję

Zauważmy, że:

a) funkcje dające w wyniku wartości innego typu niż int powinny być zadeklarowane przed użyciem;

b) funkcje zaimplementowane jako makrodefinicje mogą mieć niepożądane efekty uboczne; w rozdziale 6 w haśle *makrodefinicja* znajdują się informacje o efektach ubocznych i sposobach ich unikania;

c) TRUE i FALSE są to stałe symboliczne, które zwykle definiuje się następująco:

```
#define FALSE 0
#define TRUE 1
```

d) wymienione tutaj nazwy włączanych plików (dyrektywą #include) są poprawne dla systemu Unix i mogą się nieco różnić w innych implementacjach.

Funkcje wejścia-wyjścia

Większość implementacji udostępnia dwa rodzaje funkcji wejścia-wyjścia: niebuforowane (czyli niskiego poziomu) i buforowane. Funkcje niebuforowane, działające na deskryptorach plików (liczbach całkowitych), są bezpośrednimi wywołaniami systemu operacyjnego i, jak to wynika z określenia, nie zapewniają ani

buforowania, ani formatowania danych. Używane są głównie w programach obsługi urządzeń zewnętrznych i do przetwarzania plików nietekstowych. Funkcje buforowane działają na strumieniach (plikach tekstowych) i zapewniają buforowanie i formatowanie.

Funkcje wejścia-wyjścia niebuforowane (niskiego poziomu)

---

close zamyka plik

```
int close(deskryptor_pliku)
int deskryptor_pliku;
```

Daje w wyniku: –1, gdy wystąpił błąd.

Ponieważ otwarte pliki nie zamknięte przed zakończeniem programu zostaną zamknięte przez funkcję exit, więc nie jest niezbędne robienie tego w programie. Niemniej jednak, ze względu na możliwość załamania się systemu w najmniej odpowiedniej chwili, zaleca się zamykanie plików przed zakończeniem.

---

creat tworzy plik

```
int creat(nazwa_pliku, rodzaj_ochrony)
char *nazwa_pliku;
int rodzaj_ochrony;
```

Dołączyć pliki: <sys/types.h> <sys/stat.h>

Daje w wyniku: –1, gdy wystąpił błąd.

Funkcja creat albo tworzy nowy plik, albo otwiera istniejący i niszczy jego zawartość, to znaczy ustala jego długość na 0.

Postać argumentów istotnie zależy od implementacji.

---

eof sprawdza znacznik końca pliku

```
int eof(deskryptor_pliku)
int deskryptor_pliku;
```

Daje w wyniku: –1, gdy wystąpił błąd.

Funkcja eof daje w wyniku 1, gdy wystąpił koniec pliku i 0 w przeciwnym razie.

---

lseek zmienia bieżącą pozycję pliku

```
long lseek(deskryptor_pliku, odstęp, początek)
int deskryptor_pliku;
long odstęp;
int początek;
```

Daje w wyniku: –1L, gdy wystąpił błąd.

Funkcja lseek wymusza zmianę bieżącej pozycji pliku w zależności od wartości parametru początek w następujący sposób:

początek równy	Bieżąca pozycja ustawiona na
0	odstęp bajtów
1	obecną wartość + odstęp bajtów
2	koniec pliku + odstęp bajtów

Bieżąca pozycja może być przesunięta za koniec pliku, ale nie przed jego początek.

---

open otwiera plik

```
int open(nazwa_pliku, tryb, rodzaj_ochrony)
char *nazwa_pliku;
int tryb;
int rodzaj_ochrony;
```

Dołączyć pliki: <fcntl.h> <sys/types.h> <sys/stat.h>
Daje w wyniku: –1, gdy wystąpił błąd
Postać argumentów istotnie zależy od implementacji.

---

read czyta z pliku do bufora n bajtów

```
int read(deskryptor_pliku, bufor, n)
int deskryptor_pliku;
char *bufor;
int n;
```

Daje w wyniku: liczbę przeczytanych bajtów lub –1, gdy wystąpił błąd.

---

tell podaje wartość bieżącej pozycji w pliku

```
long tell(deskryptor_pliku)
int deskryptor_pliku;
```

Daje w wyniku: –1, gdy wystąpił błąd.

---

write zapisuje do pliku n bajtów z bufora

```
int write(deskryptor_pliku, bufor, n)
int deskryptor_pliku;
char *bufor;
int n;
```

Daje w wyniku: liczbę zapisanych bajtów lub –1, gdy wystąpił błąd.

Buforowane funkcje wejścia-wyjścia

W pliku stdio.h jest m.in. zdefiniowany wskaźnik do rekordu o nazwie FILE i stała symboliczna EOF, używana jako znacznik błędu albo końca pliku. W systemach Unix zewnętrzna zmienna całkowita errno informuje o przyczynie błędu.

---

clearerr zeruje znacznik błędu

```
int clearerr(wsk_pliku)
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>

Funkcja clearerr zeruje także znacznik końca pliku.

---

fclose zamyka plik

```
int fclose(wsk_pliku)
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>

Daje w wyniku: EOF, gdy wystąpił błąd lub 0 w przeciwnym razie.

Ponieważ pliki nie zamknięte przed zakończeniem programu zostaną zamknięte przez funkcję exit, więc nie jest niezbędne zamykanie plików w programie. Niemniej jednak jest zalecane zamykanie każdego pliku tak wcześnie, jak to tylko możliwe w celu opróżnienia bufora i zwolnienia wskaźnika FILE do ponownego użycia.

---

feof daje w wyniku TRUE, gdy koniec pliku

```
int feof(wsk_pliku)
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>

feof jest makrodefinicją.

---

ferror daje w wyniku TRUE, gdy wystąpił błąd operacji na pliku

```
int ferror(wsk_pliku)
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>

Daje w wyniku: TRUE, jeżeli ostatnie działanie na wsk_pliku (niezależnie od tego, kiedy było wykonane) spowodowało błąd.

ferror jest makrodefinicją.

---

fflush wymusza zapisanie bufora

```
int fflush(wsk_pliku)
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: EOF, gdy wystąpił błąd lub 0 w przeciwnym razie.

Funkcja fflush zapewnia, że dane są rzeczywiście zapisane do pliku. Zauważmy, że funkcje wyjścia nie piszą do pliku, lecz tylko umieszczają dane w buforze. Faktyczna operacja zapisu korzysta z funkcji systemu operacyjnego. Nie można zakładać, że w chwili załamania systemu bufor został zapisany do pliku tylko dlatego, że użyto np. funkcji fputc.

---

fgetc czyta kolejny znak z pliku

```
int fgetc(wsk_pliku)
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: kod wczytanego znaku albo EOF, jeżeli wystąpił błąd lub koniec pliku.

---

fgets czyta znaki z pliku

```
char *fgets(ciąg, n, wsk_pliku)
char *ciąg;
int n;
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: adres ciągu przy poprawnym wykonaniu albo NULL, gdy wystąpił błąd lub gdy pierwszym odczytanym znakiem jest koniec pliku.

Funkcja fgets przestaje czytać po odczytaniu n – 1 znaków lub napotkaniu znaków nowego wiersza ('\n') bądź EOF. Znak nowego wiersza jest zapamiętywany w ciągu. Po ostatnim odczytanym znaku jest dodawany znak końca ciągu ('\0').

---

fileno daje w wyniku deskryptor pliku

```
int fileno(wsk_pliku)
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>

Całkowitoliczbowe deskryptory plików są używane tylko w niebuforowanych funkcjach wejścia-wyjścia.

---

fopen otwiera plik

```
FILE *fopen(nazwa_pliku, rodzaj_dostępu)
char *nazwa_pliku;
char *rodzaj_dostępu;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: wskaźnik do rekordu FILE lub NULL, gdy wystąpił błąd.

Niektóre z dopuszczalnych wartości parametru rodzaj_dostępu: "r" – czytanie, "w" – pisanie, "a" – dopisywanie.

Rodzaj_dostępu został zadeklarowany jako ciąg znaków, ponieważ może być dłuższy niż jeden znak, gdy trzeba dostarczyć dodatkowych informacji. Działanie fopen istotnie zależy od implementacji. Programista powinien uważnie przeczytać opis tej funkcji w opisie implementacji.

---

fprintf zapisuje argumenty do pliku

```
int fprintf(wsk_pliku, format, argument...)
FILE *wsk_pliku;
char *format;
```

(argument – zmienna liczba argumentów odpowiednio do parametru format).
Dołączyć plik: <stdio.h>

Funkcja fprintf używa tego samego sposobu formatowania co printf i sprintf. Różnica między nimi polega na tym, że printf pisze do pliku stdout, fprintf do zadanego pliku, a sprintf tworzy ciąg znaków. Szczegółowy opis parametru format znajduje się na końcu tego dodatku.

---

fputc zapisuje znak do pliku

```
int fputc(c, wsk_pliku)
char *c;
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: 0 przy poprawnym działaniu lub EOF, gdy wystąpił błąd.

Parametr fputc jest typu char, podczas gdy analogiczna funkcja wejścia fgetc daje w wyniku wczytany znak w postaci liczby całkowitej typu int.

fputc jest funkcją, a putc i putchar są makrodefinicjami.

---

fputs zapisuje ciąg znaków do pliku

```
int fputs(ciąg, wsk_pliku)
char *ciąg;
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku 0 przy poprawnym działaniu lub EOF, gdy wystąpił błąd.

---

fread czyta bloki danych z pliku do tablicy znaków

```
int fread(ciąg, n1, n2, wsk_pliku)
char *ciąg;
int n1;        /* liczba bajtów w każdym bloku */
```

```
int n2; /* liczba bloków do odczytania */
FILE * wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: liczbę odczytanych bloków przy poprawnym działaniu lub EOF, gdy wystąpił błąd.

---

freopen zamyka plik i otwiera inny

```
FILE *freopen(nazwa_pliku, rodzaj_dostępu, wsk_pliku)
char *nazwa_pliku;
char *rodzaj_dostępu;
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: EOF, gdy wystąpił błąd.

Funkcja freopen zamyka plik skojarzony ze wsk_pliku i otwiera plik o nazwie nazwa_pliku. Dozwolone wartości parametru rodzaj_dostępu zawarto w opisie funkcji fopen.

---

fscanf czyta argumenty z pliku

```
int fscanf(wsk_pliku, format, argument..)
FILE *wsk_pliku;
char *format;
```

(argument – argumenty w liczbie odpowiedniej do parametru format).
Dołączyć plik: <stdio.h>
Daje w wyniku: liczbę odczytanych argumentów lub EOF, gdy wystąpił błąd.

Funkcja fscanf korzysta z tego samego mechanizmu formatowania co scanf i sscanf. Różnica między tymi funkcjami polega na tym, że scanf czyta z pliku stdin, fscanf z zadanego pliku, a sscanf przetwarza ciąg znaków. Szczegółowy opis parametru format znajduje się na końcu tego dodatku.

---

fseek zmienia bieżącą pozycję pliku

```
int fseek(wsk_pliku, odstęp, początek)
FILE *wsk_pliku;
int odstęp;
int początek;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: 0 przy poprawnym działaniu, inną liczbę, gdy wystąpił błąd.

Funkcja fseek przesuwa bieżącą pozycję pliku w zależności od parametru początek w następujący sposób:

początek równy	Bieżąca pozycja ustawiona na
0	odstęp bajtów
1	bieżąca pozycja + odstęp bajtów
2	koniec pliku + odstęp bajtów

Bieżąca pozycja może być przesunięta za koniec pliku, lecz nie przed jego początek.

---

ftell daje w wyniku przesunięcie od początku pliku

```
long ftell(wsk_pliku)
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: presunięcie od początku pliku lub –1L, gdy wystąpił błąd.

---

fwrite zapisuje bloki danych do pliku

```
int fwrite(ciąg, n1, n2, wsk_pliku)
char *ciąg;
int n1;              /* liczba bajtów w każdym bloku */
int n2;              /* liczba bloków */
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: liczbę zapisanych bloków lub NULL, gdy wystąpił błąd.

Funkcja fwrite zapisuje n2 bloków danych z ciągu do pliku wskazywanego przez wsk_pliku. Zauważmy, że ciąg nie musi być tablicą znakową, lecz raczej wskaźnikiem do tablicy "czegoś".

---

getc czyta kolejny znak z pliku

```
int getc(wsk_pliku)
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: kod wczytanego znaku albo EOF, gdy wystąpił błąd lub koniec pliku.

getc jest makrodefinicją.

---

getchar czyta kolejny znak z pliku stdin

```
int getchar()
```

Dołączyć plik: <stdio.h>
Daje w wyniku: kod wczytanego znaku albo EOF, gdy wystąpił błąd lub koniec pliku.

getchar jest makrodefinicją.

---

gets czyta znaki z pliku stdin do ciągu znaków

```
char *gets(ciąg)
char *ciąg;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: adres ciągu przy poprawnym działaniu albo NULL, gdy wystąpił błąd lub gdy pierwszym wczytanym znakiem jest znak końca pliku.

Funkcja gets kończy czytanie po napotkaniu końca pliku albo znaku nowego wiersza ('\n'). Znak nowego wiersza nie jest zapamiętywany w ciągu, jak w przypadku użycia funkcji fgets.

---

getw czyta słowo (liczbę całkowitą) z pliku

```
int getw(wsk_pliku)
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: wczytaną liczbę całkowitą lub EOF, gdy wystąpił błąd. Ze względu na to, że EOF jest dopuszczalną liczbą całkowitą, należy użyć feof i ferror, aby sprawdzić czy wystąpił błąd.

Pliki odczytywane przez getw są nieprzenośne, ponieważ zarówno rozmiar liczby całkowitej jak i kolejność, w jakiej są zapisywane bajty, zależą od komputera.

---

printf zapisuje argumenty do pliku

```
int printf(format, argument...)
char *format;
```

(argument – liczba argumentów odpowiednia do parametru format).
Dołączyć plik: <stdio.h>
Daje w wyniku: 0 przy poprawnym działaniu lub EOF, gdy wystąpił błąd.

Funkcja printf jest podobna do fprintf, z wyjątkiem tego, że pisze do pliku stdout.

---

putc zapisuje znak do pliku

```
int putc(c, wsk_pliku)
char c;
FILE *wsk_pliku;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: kod zapisanego znaku lub EOF, gdy wystąpił błąd.

putc jest makrodefinicją – odpowiednikiem fputc.

---

putchar zapisuje znak do pliku stdout

```
int putchar(c)
char c;
```

Dołączyć plik: < stdio.h >
Daje w wyniku: kod zapisanego znaku lub EOF, gdy wystąpił błąd.
putchar jest makrodefinicją.

---

puts zapisuje ciąg znaków do pliku stdout

```
int puts(ciąg)
char *ciąg;
```

Dołączyć plik: < stdio.h >

---

putw zapisuje słowo (liczbę całkowitą) do pliku

```
int putw(i, wsk_pliku)
int i;
FILE *wsk_pliku;
```

Dołączyć plik: < stdio.h >
Daje w wyniku: zapisaną liczbę całkowitą lub EOF, gdy wystąpił błąd. Ze względu na to, że EOF jest dopuszczalną liczbą całkowitą, należy użyć feof i ferror, aby sprawdzić czy wystąpił błąd.

Pliki zapisane przez putw są nieprzenośne, ponieważ zarówno rozmiar liczby całkowitej, jak i kolejność zapisywania bajtów zależy od komputera.

---

scanf czyta argument z pliku stdin

```
int scanf(format, argument...)
char *format;
```

(argument – argumenty w liczbie odpowiedniej do parametru format).
Dołączyć plik: < stdio.h >
Daje w wyniku: liczbę odczytanych argumentów.

Funkcja scanf używa tego samego mechanizmu formatowania co fscanf i sscanf. Różnica między tymi funkcjami polega na tym, że scanf czyta z pliku stdin, fscanf z zadanego pliku, a sscanf przetwarza ciąg znaków.

---

sscanf czyta argument z ciągu znaków

```
int sscanf(ciąg, format, argument...)
char *ciąg;
char *format;
```

(argument – argumenty w liczbie odpowiedniej do parametru format).

Dołączyć plik: < stdio.h >
Daje w wyniku: liczbę odczytanych argumentów.

Funkcja sscanf używa tego samego mechanizmu formatowania co fscanf i scanf. Różnica między tymi funkcjami polega na tym, że scanf czyta z pliku stdin, fscanf z zadanego pliku, a sscanf przetwarza ciąg znaków.

---

tmpfile tworzy plik tymczasowy i otwiera go do zapisywania

```
FILE *tmpfile()
```

Dołączyć plik: < stdio.h >
Daje w wyniku: wskaźnik do rekordu typu FILE albo NULL, gdy wystąpił błąd.

Plik tymczasowy jest usuwany przy zakończeniu programu.

---

ungetc powoduje wycofanie znaku do pliku, tak że następne wykonanie getc da ponownie ten sam znak

```
int ungetc(c, wsk_pliku)
int c;
FILE *wsk_pliku;
```

Dołączyć plik: < stdio.h >
Daje w wyniku: c przy poprawnym działaniu lub EOF, gdy wystąpił błąd.

Tylko jeden znak może być wycofany bez ponownego odczytu. EOF nie może być wycofany. Funkcja fseek anuluje efekty działania ungetc.

---

unlink usuwa plik

```
int unlink(nazwa_pliku)
char *nazwa_pliku;
```

Dołączyć plik: < stdio.h >
Daje w wyniku: –1, gdy wystąpił błąd.

Zauważmy, że parametrem jest tu nazwa pliku, a nie wskaźnik do rekordu FILE. Oznacza to, że nie jest konieczne otwarcie pliku przed jego usunięciem.

Funkcje przetwarzania ciągów znaków

Żadna ze standardowych funkcji przetwarzania ciągów znaków nie przydziela pamięci na wyniki, lecz zakłada, że uprzednio zrobił to programista. Jeżeli tak się nie stało, to funkcje te zapiszą wyniki na tym, co im stanie na drodze.

W dodatku B znajduje się kod źródłowy kilku dodatkowych funkcji przetwarzania ciągów, które prawdopodobnie mogą się okazać przydatne.

---

index znajduje pierwsze wystąpienie znaku w ciągu

```
char *index(ciąg, c)
char *ciąg;
char c;
```

Daje w wyniku: położenie znaku w ciągu albo NULL, gdy znak nie został znaleziony.

---

memchr znajduje pierwsze wystąpienie znaku w początkowych n znakach ciągu

```
char *memchr(ciąg, c, n)
char *ciąg;
char c;
int n;
```

Dołączyć plik: <memory.h>
Daje w wyniku: NULL w razie niepowodzenia.

---

memcmp porównuje n początkowych znaków ciągów

```
int memcmp(ciąg1, ciąg2, n)
char *ciąg1;
char *ciąg2;
int n;
```

Dołączyć plik: <memory.h>
Daje w wyniku: liczbę mniejszą niż 0, jeżeli leksykograficznie ciąg1 < ciąg2, 0, jeżeli ciąg1 = ciąg2 i liczbę większą niż 0, jeżeli ciąg1 > ciąg2.

---

memccpy kopiuje kolejne znaki z jednego ciągu do drugiego, aż do skopiowania zadanego znaku albo zadanej liczby znaków

```
char *memccpy(ciąg1, ciąg2, c, n)
char *ciąg1;
char *ciąg2;
char c;
int n;
```

Dołączyć plik: <memory.h>

Funkcja memccpy kopiuje ciąg2 do ciągu1, aż do skopiowania znaku c albo do skopiowania n znaków. Argumenty są w tradycyjnej, "odwrotnej" kolejności.

---

memcpy kopiuje n znaków z jednego ciągu do drugiego

```
char *memcpy(ciąg1, ciąg2, n)
char *ciąg1;
char *ciąg2;
int n;
```

Dołączyć plik: <memory.h>

Funkcja memcpy kopiuje ciąg2 do ciągu1, aż do skopiowania n znaków. Argumenty są podawane w tradycyjnej, "odwrotnej" kolejności.

---

memset ustawia n znaków na zadaną wartość

```
char *memset(ciąg, c, n)
char *ciąg;
char c;
int n;
```

Dołączyć plik: <memory.h>

Uwaga: ta funkcja jest dostępna nie we wszystkich implementacjach języka C. Kod źródłowy tej funkcji podano w dodatku B.

---

rindex znajduje ostatnie wystąpienie znaku w ciągu

```
char *rindex(ciąg, c)
char *ciąg;
char c;
```

Daje w wyniku: położenie znaku w ciągu lub NULL, gdy znak nie został znaleziony.

---

sprintf "zapisuje" argumenty do ciągu

```
int sprintf(ciąg, format, argument...)
char *ciąg;
char *format;
```

(argument – argumenty w liczbie odpowiedniej do parametru format).

Dołączyć plik: <stdio.h>

Funkcja sprintf używa tego samego mechanizmu formatowania co fprintf i printf. Różnica między tymi funkcjami polega na tym, że printf pisze do pliku stdout, fprintf do zadanego pliku, a sprintf tworzy ciąg znaków.

---

strcat łączy dwa ciągi

```
char *strcat(ciąg1, ciąg2)
char *ciąg1;
char *ciąg2;
```

Dołączyć plik: <string.h>

Wynik otrzymuje się w ciągu ciąg1

Funkcja strcat dołącza ciąg2 na koniec ciągu ciąg1.

---

strchr znajduje pierwsze wystąpienie znaku w ciągu

```
char *strchr(ciąg, c)
char c;
char *ciąg;
```

Dołączyć plik: <string.h>
Daje w wyniku: wskazanie pierwszego wystąpienia znaku c lub NULL, gdy c nie występuje w ciągu.
Uwaga: ta funkcja jest dostępna nie we wszystkich implementacjach języka C. Kod źródłowy tej funkcji znajduje się w dodatku B.

---

strcmp porównuje dwa ciągi

```
int strcmp(ciąg1, ciąg2)
char *ciąg1;
char *ciąg2;
```

Dołączyć plik: <string.h>
Daje w wyniku: liczbę mniejszą niż 0, jeżeli leksykograficznie ciąg1 < ciąg2, 0, jeżeli ciąg1 = ciąg2 i liczbę większą niż 0, jeżeli ciąg1 > ciąg2..

---

strcpy kopiuje jeden ciąg do drugiego

```
char *strcpy(ciąg1, ciąg2)
char *ciąg1;
char *ciąg2;
```

Dołączyć plik: <string.h>
Wynik otrzymuje się w ciągu ciąg1

Funkcja strcpy kopiuje zawartość ciągu2 do ciągu1. Argumenty są podane w tradycyjnej, "odwrotnej" kolejności.

---

strlen daje w wyniku długość ciągu

```
int strlen (ciąg)
char *ciąg;
```

Dołączyć plik: <string.h>

Końcowy znak '\0' nie jest liczony.

---

strncat dołącza znaki jednego ciągu do drugiego

```
char *strncat(ciąg1, ciąg2, n)
char *ciąg1;
char *ciąg2;
int n;
```

Dołączyć plik: <string.h>

Wynik otrzymuje się w ciągu ciąg1.

Funkcja ta dołącza co najwyżej n początkowych znaków ciągu2 do ciągu1.

---

strncmp porównuje znaki w ciągach

```
int strncmp(ciąg1, ciąg2, n)
char *ciąg1;
char *ciąg2;
int n;
```

Dołączyć plik: < string.h >
Daje w wyniku: liczbę mniejszą niż 0, jeżeli leksykograficznie ciąg1 < ciąg2, 0, jeżeli ciąg1 = ciąg2 i liczbę większą niż 0, jeżeli ciąg1 > ciąg2.

Funkcja strncmp porównuje co najwyżej n początkowych znaków każdego z ciągów.

---

strncpy kopiuje znaki z jednego ciągu do drugiego

```
char *strncpy(ciąg1, ciąg2, n)
char *ciąg1;
char *ciąg2;
int n;
```

Dołączyć plik: < string.h >
Wynik otrzymuje się w ciągu ciąg1.

Funkcja strncpy kopiuje co najwyżej n początkowych znaków ciągu2 do ciągu1. Argumenty są podane w tradycyjnej, "odwrotnej" kolejności.

---

strrchr znajduje ostatnie wystąpienie znaku w ciągu

```
char *strrchr(ciąg, c)
char *ciąg;
char c;
```

Dołączyć plik: < string.h >
Daje w wyniku: wskaźnik do miejsca ostatniego wystąpienia c lub NULL, jeżeli c nie występuje w ciągu.
Uwaga: ta funkcja jest dostępna nie we wszystkich implementacjach języka C. Jej kod źródłowy podano w dodatku B.

Funkcje przetwarzania znaków

Większość funkcji podanych w tej części to makrodefinicje. Wartości szesnastkowe są poprawne tylko dla zestawu znaków ASCII.

---

isalnum sprawdza czy znak jest alfanumeryczny

```
int isalnum(c)
char c;
```

Dołączyć plik: < ctype.h >
Daje w wyniku: TRUE, jeżeli c jest znakiem alfanumerycznym, tj. 'A'..'Z', 'a'..'z' lub '0'..'9'.

---

isalpha sprawdza czy znak jest literą

```
int isalpha(c)
char c;
```

Dołączyć plik: < ctype.h >
Daje w wyniku: TRUE, jeżeli c jest znakiem alfabetycznym, to znaczy 'A'..'Z' lub 'a'..'z'.

---

isascii sprawdza czy znak należy do kodu ASCII

```
int isascii(c)
char c;
```

Dołączyć plik: < ctype.h >
Daje w wyniku: TRUE, gdy c jest znakiem ASCII, tj. o kodzie mniejszym niż 0x80 (128).

---

iscntrl sprawdza czy znak jest znakiem sterującym

```
int iscntrl(c)
char c;
```

Dołączyć plik: < ctype.h >
Daje w wyniku: TRUE, jeżeli c jest znakiem sterującym, to znaczy o kodzie mniejszym niż 0x20 (32) albo równym 0x7F (127).

---

isdigit sprawdza czy znak jest cyfrą

```
int isdigit(c)
char c;
```

Dołączyć plik: < ctype.h >
Daje w wyniku: TRUE, jeżeli c jest cyfrą, to znaczy '0'..'9'.

---

isgraph sprawdza czy znak jest znakiem graficznym

```
int isgraph(c)
char c;
```

Dołączyć plik: <ctype.h>
Daje w wyniku: TRUE, jeżeli c jest znakiem graficznym, to znaczy o kodzie zawartym między 0x21 (128) a 0x7E (126).

---

islower sprawdza czy znak jest małą literą

```
int islower(c)
char c;
```

Dołączyć plik: <ctype.h>
Daje w wyniku: TRUE, jeżeli c jest małą literą, to znaczy 'a'..'z'.

---

isprint sprawdza czy znak daje się wydrukować

```
int isprint(c)
char c;
```

Dołączyć plik: <ctype.h>
Daje w wyniku: TRUE, jeżeli c jest znakiem dającym się wydrukować, to znaczy o kodzie zawartym między 0x20 (32) a 0x7E (126).

---

ispunct sprawdza czy znak jest znakiem interpunkcyjnym

```
int ispunct(c)
char c;
```

Dołączyć plik: <ctype.h>
Daje w wyniku: TRUE, jeżeli c jest znakiem interpunkcyjnym (lub inaczej nie sterującym i nie alfanumerycznym).

---

isspace sprawdza czy znak jest zdefiniowany jako odstęp

```
int isspace(c)
char c;
```

Dołączyć plik: <ctype.h>
Daje w wyniku: TRUE, jeżeli c jest znakiem odstępu (spacją, znakiem nowego wiersza, nowej strony, powrotu karetki albo tabulatorem poziomym lub pionowym).

---

isupper sprawdza czy znak jest wielką literą

```
int isupper(c)
char c;
```

Dołączyć plik: <ctype.h>
Daje w wyniku: TRUE, jeżeli c jest wielką literą, to znaczy 'A'..'Z'.

---

toascii przekształca znak na znak kodu ASCII

int toascii(c) char c;

Dołączyć plik: <ctype.h>

Funkcja toascii zeruje wszystkie bity c z wyjątkiem siedmiu najmniej znaczących.

---

tolower zamienia wielką literę na małą

int tolower(c)
char c;

Dołączyć plik: <ctype.h>
Daje w wyniku: małą literę – odpowiednik c.

Należy użyć isupper, aby sprawdzić, czy c jest rzeczywiście wielką literą.

---

toupper zamienia małą literę na wielką

int toupper(c) char c;

Dołączyć plik: <ctype.h>
Daje w wyniku: wielką literę – odpowiednik c.

Należy użyć islower, aby sprawdzić, czy c jest rzeczywiście małą literą.

Funkcje matematyczne

W pliku math.h jest zdefiniowana stała symboliczna HUGE określająca wartość, która nie może pojawić się w toku normalnych obliczeń. Przykładowo może to być 1.701411733192644270E38 albo 99E99 (nieskończoność według definicji IEEE). Stała ta jest często używana do sygnalizowania błędu.

Argumenty funkcji trygonometrycznych są wyrażone w radianach.

---

abs wartość bezwzględna

int abs(n)
int n;

Daje w wyniku: wartość bezwzględną n.

---

acos arcus cosinus

double acos(d)
double d;

Dołączyć plik: <math.h>

Daje w wyniku: 0, gdy wystąpił błąd.

---

asin arcus sinus

```
double asin(d)
double d;
```

Dołączyć plik: <math.h>
Daje w wyniku: 0, gdy wystąpił błąd.

---

atan arcus tangens

```
double atan(d)
double d;
```

Dołączyć plik: <math.h>

---

atan2 arcus tangens d1/d2

```
double atan2(d1, d2)
double d1;
double d2;
```

Dołączyć plik: <math.h>
Daje w wyniku: HUGE, gdy wystąpił błąd.
Należy upewnić się, czy d2 nie jest zerem przed użyciem atan2.

---

atoi przekształca ciąg znaków na liczbę całkowitą

```
int atoi(ciąg)
char *ciąg;
```

Daje w wyniku: wartość liczby zapisanej w ciągu.

---

atof przekształca ciąg znaków na liczbę rzeczywistą

```
double atof(ciąg)
char *ciąg;
```

Dołączyć plik: <math.h>
Daje w wyniku: HUGE lub 0, gdy wystąpił błąd.
Ciąg powinien być zapisany w notacji naukowej (patrz hasło *stała* w rozdz. 6).

---

atol przekształca ciąg znaków na liczbę całkowitą o podwójnej precyzji

```
long atol(ciąg)
char *ciąg;
```

Daje w wyniku: 0, gdy wystąpił błąd.

ceil daje w wyniku najmniejszą liczbę całkowitą nie mniejszą niż d

```
double ceil(d)
double d;
```

Dołączyć plik: <math.h>

---

cos cosinus

```
double cos(d)
double d;
```

Dołączyć plik: <math.h>

---

cosh cosinus hiperboliczny

```
double cosh(d)
double d;
```

Dołączyć plik: <math.h>

---

exp e do potęgi d

```
double exp(d)
double d;
```

Dołączyć plik: <math.h>
Daje w wyniku: HUGE, gdy wystąpił błąd.

---

fabs wartość bezwzględna d

```
double fabs(d)
double d;
```

Dołączyć plik: <math.h>

---

floor największa liczba całkowita nie większa niż d

```
double floor(d)
double d;
```

Dołączyć plik: <math.h>

---

fmod reszta z dzielenia d1 przez d2

```
double fmod(d1, d2)
double d1;
double d2;
```

Dołączyć plik: <math.h>

Przed użyciem fmod należy upewnić się, że d2 nie jest zerem.

---

log logarytm naturalny

```
double log(d)
double d;
```

Dołączyć plik: <math.h>
Daje w wyniku: 0, gdy wystąpił błąd.

---

log10 logarytm dziesiętny

```
double log10(d)
double d;
```

Dołączyć plik: <math.h>
Daje w wyniku: 0, gdy wystąpił błąd.

---

pow potęga

```
double pow(d1, d2)
double d1;
double d2;
```

Dołączyć plik: <math.h>
Daje w wyniku: HUGE lub 0, gdy wystąpił błąd.

Funkcja pow daje w wyniku d1 podniesione do potęgi d2. W niektórych implementacjach ta funkcja jest nazwana power.

---

rand liczba pseudolosowa

```
int rand()
```

---

random liczba pseudolosowa

```
int random()
```

Zakłada się, że random jest lepszym generatorem liczb pseudolosowych niż rand.

---

sin sinus

```
double sin(d)
double d;
```

Dołączyć plik: <math.h>

---

sinh sinus hiperboliczny

```
double sinh(d)
double d;
```

Dołączyć plik: <math.h>

---

sqrt pierwiastek kwadratowy

```
double sqrt(d)
double d;
```

Dołączyć plik: <math.h>
Daje w wyniku: 0, gdy wystąpił błąd.

---

srand ustawia generator liczb pseudolosowych

```
void srand(u)
unsigned int u;
```

---

tan tangens

```
double tan(d)
double d;
```

Dołączyć plik: <math.h>
Daje w wyniku: HUGE, gdy wystąpił błąd.

---

tanh tangens hiperboliczny

```
double tanh(d)
double d;
```

Dołączyć plik: <math.h>
Daje w wyniku: HUGE, gdy wystąpił błąd.

Funkcje przydzielania i zwalniania pamięci

Te funkcje są używane do przydzielania pamięci na rekordy lub tablice wówczas, gdy zapotrzebowanie na pamięć jest dynamiczne, czyli nie jest znane w trakcie kompilacji, lecz dopiero podczas wykonania. Pamięć przydzielona przez jedną z funkcji alloc może być oddana systemowi operacyjnemu (do ponownego użycia) za pomocą funkcji free.

Stała symboliczna NULL, zdefiniowana w pliku stdio.h, jest używana do wskazywania błędów

---

calloc przydziela pamięć

```
char *calloc(u1, u2)
unsigned int u1;
unsigned int u2
```

Daje w wyniku: wskaźnik do przydzielonej pamięci lub NULL w razie niepowodzenia.

Funkcja calloc przydziela pamięć na u1 elementów o rozmiarze u2 bajtów każdy i wypełnia ją zerami.

---

free zwalnia pamięć

```
void free(ciąg)
char *ciąg;
```

Daje w wyniku: NULL w razie niepowodzenia

Uwaga: ciąg powinien być uprzednio przydzielony przez calloc lub malloc.

---

malloc przydziela pamięć

```
char *malloc(u)
unsigned u;
```

Daje w wyniku: wskaźnik do przydzielonej pamięci albo NULL w razie niepowodzenia.

Funkcja malloc przydziela pamięć dla u elementów. Jeżeli trzeba przydzielić pamięć dla zmiennych innego typu niż char, to jest konieczna konwersja rezultatu funkcji jak w poniższych przykładach

```
wsk_do_liczby_typu_int = (int*)malloc(sizeof(int));
return ((struct węzeł *)malloc(sizeof(struct węzeł)));
```

---

realloc zmienia przydział pamięci

```
char *realloc(ciąg, u)
char *ciąg;
unsigned int u;
```

Daje w wyniku wskaźnik do przydzielonej pamięci lub NULL w razie niepowodzenia.

Funkcja realloc zmienia rozmiar przydzielonej pamięci wskazywanej przez ciąg do u bajtów. Ciąg powinien być uprzednio przydzielony przez calloc lub malloc.

Funkcje daty i czasu

Te funkcje nie są zaimplementowane w niektórych komputerach. Komputer musi mieć zegar czasu rzeczywistego, do którego ma dostęp system operacyjny. Następujący rekord (zdefiniowany w pliku time.h) jest używany przez liczne funkcje z tej części:

```
struct tm;
{
   int tm_sec;    /* sekundy */
   int tm_min;    /* minuty */
   int tm_hour;   /* godziny */
   int tm_mday;   /* dzień miesiąca (1–31) */
   int tm_year;   /* rok (bieżący rok – 1900) */
   int tm_wday;   /* dzień tygodnia (0–6) */
   int tm_yday;   /* dzień roku (0 –365) */
   int tm_isdst;  /* nie–zero, jeżeli obowiązuje czas letni */
};
```

---

clock mierzy czas

long clock()

Funkcja clock daje w wyniku liczbę mikrosekund czasu pracy procesora, jaki upłynął od ostatniego wywołania clock.

---

ctime przekształca czas na ciąg znaków

char *ctime(l);
long *l;

Dołączyć plik: <time.h>

Funkcja ctime pozwala przekształcić wartość l, otrzymaną z funkcji time na postać znakową.

---

gmtime przekształca czas na czas GMT (średni czas Greenwich)

struct tm *gmtime()
long *l;

Dołączyć plik: <time.h>
Daje w wyniku: wskaźnik do rekordu tm.

---

localtime przekształca czas na czas lokalny

struct tm *localtime(l)
long *l;

Dołączyć plik: <time.h>
Daje w wyniku: wskaźnik do rekordu tm.

---

time dajc w wyniku datę i czas

```
long time(l)
long *l;
```

Dołączyć plik: <time.h>
Daje w wyniku: datę i czas w sekundach.

Jeżeli l nie jest zerem, to rezultat funkcji jest także zapamiętywany w miejscu pamięci wskazywanym przez l.

Funkcje różne

---

system wykonuje polecenie systemu operacyjnego zawarte w ciągu znaków

```
system(ciąg)
char *ciąg;
```

Dołączyć plik: <stdio.h>
Daje w wyniku: wartość zwracaną przez wykonywane polecenie albo –1, gdy polecenie nie zostało wykonane.

Ciąg powinien zawierać poprawne polecenie odpowiedniego systemu operacyjnego. Wartość otrzymywana w wyniku w znacznym stopniu zależy od systemu.

---

exit kończy wykonanie programu

```
exit(kod_powrotu)
int kod_powrotu;
```

Funkcja exit kończy wykonanie programu i przekazuje systemowi operacyjnemu kod_powrotu. Wszystkie bufory są zapisywane do plików, a wszystkie otwarte pliki zamykane. Zwykle kod_powrotu równy zero oznacza poprawne wykonanie.

_exit kończy wykonanie programu

```
_exit(kod_powrotu)
int kod_powrotu;
```

Funkcja _exit kończy wykonanie programu i przekazuje systemowi operacyjnemu kod_powrotu. Bufory nie są zapisywane do plików, a otwarte pliki nie są zamykane. Zwykle kod_powrotu równy zero oznacza poprawne wykonanie. Funkcja _exit nie powinna być używana w normalnych programach – należy korzystać z exit.

---

qsort sortuje

```
qsort(baza, n, rozmiar, funkcja_porównująca)
char *baza;
```

```
int n;
int rozmiar;
int *funkcja_porównująca();
```

Funkcja qsort sortuje w pamięci tablicę n elementów, o wielkości rozmiar bajtów każdy, wskazywaną przez wskaźnik baza. Funkcja_porównująca jest nazwą funkcji dostarczonej przez programistę (musi ona być zadeklarowana w funkcji wywołującej qsort). Parametrami tej funkcji są dwa wskaźniki do porównywanych elementów, a rezultatem liczba całkowita: mniejsza niż 0, jeżeli pierwszy element jest mniejszy niż drugi, większa niż 0, jeżeli pierwszy element jest większy niż drugi, oraz równa 0, jeżeli oba elementy są równe.

Format tekstu sterującego funkcjami printf

Takie same teksty sterujące są używane przez funkcje printf, fprintf i sprintf. Tekst sterujący jest ciągiem znaków, w którym argumenty mają następującą postać:

% *wskaźniki rozmiar precyzja* l *typ*

Wskaźniki mogą przyjmować wartości:

–	wyrównaj lewostronnie;
+	wartość ma być poprzedzona znakiem + lub – ;
<spacja>	wartości dodatnie mają być poprzedzone znakiem spacji
#	wartości ósemkowe mają być poprzedzone znakiem 0, szesnastkowe 0x; lub 0X, liczby zmiennopozycyjne muszą zawierać kropkę, końcowe zera dla typów G i g muszą pozostać;

rozmiar oznacza minimalną szerokość pola (* oznacza, że następny argument printf podaje rozmiar)

precyzja oznacza:
dla zmiennych typu int – minimalną liczbę wyświetlanych cyfr;
dla typów e i f – liczbę cyfr po przecinku;
dla typu g – maksymalną liczbę cyfr znaczących;
dla typu s – maksymalną liczbę znaków;
(* oznacza, że następny argument printf podaje precyzję);

l oznacza, że argument jest typu long – *typ* powinien więc być d, o ,u, x lub X

typ może przyjmować wartości:

d	int
u	unsigned int
o	int przekształcony na postać ósemkową
x	int przekształcony na postać szesnaskową, przy użyciu liter od "a" do "f"
X	int przekształcony na postać szesnastkową, przy użyciu liter od "A" do "F"
f	float

e float w zapisie naukowym
E float w zapisie naukowym, z użyciem "E" zamiast "e"
g float z użyciem specyfikacji "f" lub "e", w zależności od tego, która daje krótszy tekst bez pogorszenia dokładności
G float z użyciem użyta zostanie specyfikacji "f" lub "E", w zależności od tego, która daje krótszy tekst
c char
s ciąg znaków
% znak %

Każdy znak ciągu sterującego nie poprzedzony znakiem % będzie skopiowany do wyjścia funkcji.

PRZYKŁAD

```
int i = 10;
char *ciąg = "abcdef";
printf("\ni = %2x, s = \"%s\"",i, ciąg);
```

spowoduje wypisanie:

```
i = a, s = "abcdef"
```

Bardziej złożony sposób użycia printf:

```
printf("%c%s%s",
(k ? '\n':' '),
(k ? itoa(j,buf1,10):""),
(k ? itoa(j,buf2,10):""));
```

Format tekstu sterującego funkcjami scanf

Takie same ciągi sterujące są używane przez funkcje scanf, fscanf i sscanf. Ciąg sterujący jest ciągiem znaków, w którym argumenty mają następującą postać:

%* *rozmiar* l h *typ*

Znak * oznacza, że wśród argumentów nie ma odpowiedniego wskaźnika;

rozmiar oznacza maksymalną szerokość pola; (* oznacza, że następny argument scanf podaje *rozmiar*);

l oznacza, że wartość ma być zapamiętana jako long int albo double;

h oznacza, że wartość ma być zapamiętana jako short int;

typ może przyjmować wartości (typ odpowiedniego argumentu podano w nawiasach):

d int (int)
u unsigned int (unsigned int)
o int w postaci ósemkowej (int)
x int w postaci szesnastkowej zapisanej przy użyciu liter od "a" do "f" (int)

f	float (float)
e	float (float)
g	float (float)
c	char (char)
s	ciąg zakończony znakiem zdefiniowanym jako odstęp (tablica elementów typu char)
[...]	ciąg zakończony dowolnym znakiem nie zawartym w nawiasach (tablica elementów typu char)
[^...]	ciąg zakończony jednym ze znaków zawartych w nawiasach (tablica elementów typu char)
%	znak % (nie ma przypisania)

Każdy ze znaków nie poprzedzonych znakiem % musi dokładnie pasować do znaków wejściowych. Spacja w ciągu sterującym pasuje do dowolnej liczby następujących po sobie spacji w strumieniu wejściowym.

PRZYKŁAD

```
int i;
char *ciąg;
scanf("%d %s", i, ciąg);
```

pasuje do

```
16 bąbli
```

Dodatek B. Funkcje przetwarzania ciągów znaków

```
/****************************************************************************
FUNKCJE PRZETWARZANIA CIĄGÓW ZNAKÓW
****************************************************************************/
#include <stdio.h>
#include <ctype.h>
#define EOS '\0'          /* koniec ciągu znaków */
#define SUCCESS 0
#define FAILURE -1

/****************************************************************************
FUNKCJA strchr daje w wyniku wskaźnik do pierwszego wystąpienia znaku c
w ciągu string albo NULL, jeżeli c nie występuje.
****************************************************************************/
char *strchr(string, c)
char *string;
char c;
{
    while(*string)
      {
        if (*string == c)
          return string;
        string++;
      }
    return NULL;
}

/****************************************************************************
FUNKCJA strrchr daje w wyniku wskaźnik do ostatniego wystąpienia znaku c
w ciągu string albo NULL, jeżeli c nie wystąpił.
****************************************************************************/

char *strrchr(string, c)
char *string;
char c;
{
    char *end_string = string + strlen(string) - 1; /* koniec ciągu */
```

```
    while (end_string > = string)
      {
        if (*end_string = = c)
          return end_string;
          end_string- -;
      }
    return NULL;
}

/******************************************************************************
FUNKCJA strstr daje w wyniku wskaźnik do miejsca wystąpienia ciągu substring
w ciągu string; NULL, jeżeli ciąg nie został znaleziony.
******************************************************************************/

char *strstr(string, substring)
char *string;
char *substring;
{
    char *where = NULL; /* miejsce wystąpienia ciągu substring */
    int i;
    int length_substring = strlen(substring);

    if ((string = = NULL | | substring = = NULL)
    | | (length_substring < strlen(string)))
        return NULL;

    while ((where = strchr(string, *substring)))
      {
        string = where + 1;
        for (i = 0; i < length_substring; i+ +)
          {
            if (*(where + i) != *(substring + i))
              {
                where = NULL;
                break;
              }
          }

        if (where)
          break;
      }
    return where;
}

/******************************************************************************
FUNKCJA strtrunc skraca ciąg string zamieniając końcowe znaki odstępu na
znaki końca ciągu EOS. Funkcja daje w wyniku wskaźnik do ostatniego znaku
ciągu nie będącego odstępem albo wskaźnik do początku ciągu, jeżeli ten
miał długość zero.
******************************************************************************/

char *strtrunc(string)
char *string;
{
    char *end_ptr = string; /* wskaźnik na koniec ciągu */

/* szukaj końca ciągu */
```

```
    if(strlen(string))
      {
        end_ptr = string + strlen(string);

/* teraz na końcu; posuwaj się do tyłu zamieniając odstępy na EOS */

        while (isspace(*(--end_ptr)))
          *end_ptr = EOS;
      }
    return end_ptr;
}

/****************************************************************************
FUNKCJA strdel usuwa ciąg substring z wnętrza ciągu string. Jeżeli z jakiego-
kolwiek powodu usunięcie nie nastąpiło, to funkcja daje w wyniku NULL,
a w przeciwnym razie wskaźnik do pierwszego znaku za usuniętym ciągiem.
****************************************************************************/

char *strdel(substring, string)
char *substring;
char *string;
{
    char *where = NULL;
    register int i;
    register int j;
    int length_substring = strlen(substring);

    if ((string == NULL || substring == NULL)
    || (length_substring > strlen(string)))
      return NULL;

    if ((where = strstr(string, substring)) == NULL)
      return NULL;

    j = strlen(where);

/* wpisz to co zostało na końcu w miejsce usuniętego podciągu */

    for (i = 0; i < j; i++)
      *(where +i) = *(where + length_substring +i);

/* usuń "przesunięte" znaki */

    for (i = 0; i < length_substring; i++)
      *(where + j + i) = EOS;
    return where;
}



/****************************************************************************
FUNKCJA strins włącza ciąg insert do innego ciągu w zadanym miejscu. Funkcja
daje w wyniku wskaźnik do rozszerzonego ciągu lub NULL w razie niepowodzenia
****************************************************************************/

char *strins(string, insert)
char *string;
char *insert;
```

```
{
    int i;
    int length_string = strlen(string);
    int length_insert = strlen(insert);

    if ((string == NULL || insert == NULL)
    || (length_insert == 0))
      return NULL;

    if (strlen(string) == 0)  /* wstaw na końcu */
      {
        strcpy(string, insert);
        return string;
      }

    /* przesuń w prawo, aby zrobić miejsce na wstawkę */

    for (i = length_string - 1; i >= 0; i--)
        *(string + i + length_insert) = *(string + i);

    /* wstaw w opróżnione miejsce */

    for (i = 0; i < length_insert; i++)
      *(string + i) = *(insert + i);

      *(string + length_string +length_insert) = EOS;

    return string;
}
/*************************************************************************
FUNKCJA strrpl zamienia występujący w ciągu string ciąg old na ciąg new
i daje w wyniku wskaźnik do string, a w razie niepowodzenia NULL.
*************************************************************************/

char *strrpl(string, old, new)
char *string;
char *old;
char *new;
{
    char *found = NULL;

    if (string == NULL || old == NULL || new == NULL)
        return NULL;

    if (strlen(string) == 0 || strlen(old) == 0 || strlen(new) == 0)
        return NULL;

    if ((found = strdel(old, string)) == NULL)
        return NULL;
    else
      {
        if (strins(found, new) == NULL)
          return NULL;
        else
          return string;
      }
}
```

```
/*****************************************************************************
FUNKCJA strncnt zlicza wystąpienia znaku c w ciągu string
*****************************************************************************/

strcnt(string, c)
char *string;
char c;
{
    int count = 0; /*licznik*/

    while (*string)
        {
          if (*string = = c)
             count + +;

          string + +;
        }

    return count;
}
/*****************************************************************************
KONIEC FUNKCJI PRZETWARZANIA ŁAŃCUCHÓW
*****************************************************************************/
```

Dodatek C. Przykładowe programy – szyfrowanie i deszyfrowanie

Programy zawarte w tym dodatku symulują działanie maszyny szyfrującej i deszyfrującej z kołami kodowymi, zbliżonej do niemieckiej maszyny ENIGMA. Łatwiej jest wytłumaczyć działanie programu opisując działanie naśladowanej przez niego maszyny.

Na brzegu koła kodowego jest wypisanych (w naszym przykładzie) 26 liter alfabetu w przypadkowej kolejności. W maszynie jest pięć takich, różnych od siebie, kół. Są one ustawione pionowo obok siebie tak, że w okienku widać po jednej literze każdego z nich. Metoda szyfrowania jest następująca.

Litera pokazana przez pierwsze koło jest używana jako klucz do zaszyfrowania pierwszej litery tekstu jawnego. Potem koło jest obracane o jedną literę, tak że kolejna litera jest widoczna w okienku – wtedy następne koło jest używane do szyfrowania następnej litery tekstu, po czym także jest obracane o jedną literę – i tak dalej, aż do końca.

Wykonanie przez jedno z kół kompletnego obrotu powoduje obrót następnego koła o jedną literę. Dla ostatniego koła "następnym" jest pierwsze.

Metoda ta jest popularna, ponieważ umożliwia z krótkiego klucza (początkowego ustawienia kół) wygenerowanie długiego klucza. Bezpieczeństwo polega na posiadaniu kół, ich początkowym ustawieniu i algorytmie permutacji, jeżeli taki istnieje (tu nie podajemy żadnego).

Deszyfrowanie polega na ustawieniu kół w pozycję początkową i powtórzeniu powyższego algorytmu.

W tym rozdziale podano dwa programy. Pierwszy, rotor.c, tworzy plik kół kodowych wywołując funkcje generujące liczby pseudolosowe. Może się okazać konieczne dopasowanie programu do danego systemu. W najgorszym razie można napisać plik kół tak, aby koła nie powtarzały się i aby na żadnym z kół nie wystąpiła dwukrotnie ta sama litera.

Drugi program, cipher.c, pobiera tekst z konsoli, po czym szyfruje i deszyfruje go, pokazując rezultaty.

Oba programy włączają plik rotor.h. Są wywoływane z podaniem w wierszu poleceń nazwy używanego pliku kół.

```
/**************************************************************************
                        PLIK rotor.h
**************************************************************************/

#define NUM_ROTORS      5          /* liczba kół kodowych   */
#define ROTOR_SIZE      26         /* rozmiar każdego z kół */
#define MAX_LINE_SIZE   128        /* maks. długość wiersza */
#define EOS             '\0'       /* znak końca ciągu      */
#define FALSE           0
#define TRUE            1

/**************************************************************************
                        PLIK rotor.c
**************************************************************************/

#include <stdio.h>
#include <time.h>
#include "rotor.h"

#define randomize() srand((unsigned)time(NULL))
/* ustawia w przypadkowym miejscu generator liczb pseudolosowych */

/**************************************************************************
                        main
**************************************************************************/

main(argc,argv)
int argc;
char **argv;
{
    FILE *fp_out;          /* plik wyjściowy */
    FILE *fopen();

    if (argc == 1)
        printf("\nSposób użycia: rotor <plik wyjściowy>");
    else
      {
        if ((fp_out = fopen(argv[1], "w")) == NULL)
           printf("\nNie mogę otworzyć pliku %s", argv[1]);
        else
           {
              write_rotors(fp_out);
              fclose(fp_out);
           }
      }
    exit(0);
}

/**************************************************************************
FUNKCJA write_rotors zapisuje do pliku tekstowego zawartość kół kodowych
**************************************************************************/

write_rotors(fp_out)
FILE *fp_out;          /* plik wyjściowy */
{
    char letter[ROTOR_SIZE+1];     /* litery na brzegu koła */
    int num_letters = 0;           /* liczba liter */
```

```
    int rotor_number = 0;              /* numer koła */
    int found = FALSE;                 /* czy litera się powtarza? */
    int j, k;

    randomize();

    while (rotor_number < NUM_ROTORS)
      {
        while (num_letters < ROTOR_SIZE)
          {
            j = (rand() % ROTOR_SIZE) + 'A';
            found = FALSE;
            for (k = 0; k < num_letters; k++)
              {
                if(j == letter[k])
                  {
                    found = TRUE;
                    break;
                  }
              }
            if (!found)
              {
                letter[num_letters++] = j;
                letter[num_letters] = EOS;
              }
          }
        printf("\nkoło %x = \-%s\-", rotor_number, letter);

        for (k = 0; k < num_letters; k++)
          putc(letter[k], fp_out);
        putc('\n', fp_out);

        rotor_number++;
        num_letters = 0;
      }
}
/***************************************************************************
                  KONIEC PLIKU rotor.c
***************************************************************************/

/***************************************************************************
                  PLIK cipher.c
***************************************************************************/

#include <stdio.h>
#include "rotor.h"

char cltext[MAX_LINE_SIZE] = { EOS } ; /* tekst jawny */
unsigned char citext[MAX_LINE_SIZE + 1] = { EOS } ; /* zaszyfrowany */

struct                                 /* koła kodowe */
    {
        char text[ROTOR_SIZE];         /* litery na brzegu koła */
        short int letter;              /* obrócenie koła */

    } rotor[NUM_ROTORS] ;
```

```
/*****************************************************************************
                    main
*****************************************************************************/

main(argc, argv)
int argc;
char **argv;
{
    char line[MAX_LINE_SIZE];          /* wczytany wiersz */
    FILE *fp_rotor;                    /* plik kól kodowych */
    FILE *fopen();

    if (argc == 1)
        printf("\nSposób użycia: <plik cipher kół>");
    else
      {
        if((fp_rotor = fopen(argv[1], "r")) == NULL)
          printf("nie mogę otworzyć %s", argv[1]);
        else
          {
            read_rotors(fp_rotor);
            while (fgets(line, MAX_LINE_SIZE, stdin))
              {
                encipher(line);
                decipher(citext);
                printf("\n");
              }
          }
      }
    exit(0);
}
/*****************************************************************************
    FUNKCJA encipher szyfruje jeden wiersz tekstu jawnego
*****************************************************************************/

encipher(line)
char *line;
{
    unsigned char *citext_ptr = citext + 1; /* wskaźnik do tekstu zaszyfrowanego */
    int i;
    int rotor_number = 0;    /* numer koła */

    printf("\ntekst jawny: \"%s\" ", line);
    citext[0]=0;

    for(i = 0; i < NUM_ROTORS; i++)
        rotor[i].letter = 0;

    while (*line)
      {
        if(rotor_number == NUM_ROTORS)
          rotor_number = 0;

        *(citext_ptr++) = *line + (rotor[rotor_number].text[rotor[rotor_number].letter]);
        citext[0]++;
```

```
            turn_rotor(rotor_number++);
            line++;
          }

        printf("\ntekst zaszyfrowany: ");

        for (i = 0; i <= citext[0]; i++)
            printf("%02x ", citext[i]);
}

/****************************************************************************
      FUNKCJA decipher deszyfruje jeden wiersz tekstu zaszyfrowanego
****************************************************************************/

decipher(citext)
unsigned char *citext;
{
      char *cltext_ptr = cltext;            /* wskaźnik do tekstu jawnego */
      unsigned char *citext_ptr = citext + 1;  /* do tekstu zaszyfrowanego*/
      int i;
      int length = (int) *citext; /* długość tekstu zaszyfrowanego */
      int rotor_number = 0;                 /* numer koła */

      *cltext_ptr = EOS;

      for (i = 0; i < NUM_ROTORS; i++)
          rotor[i].letter = 0;

      for (i = 1; i <= length; i++)
        {
          if (rotor_number == NUM_ROTORS)
             rotor_number = 0;

          *(cltext_ptr++) = *citext_ptr -
             (rotor[rotor_number].text[rotor[rotor_number].letter]);

          *cltext_ptr = EOS;
          citext_ptr++;

          turn_rotor(rotor_number++);
        }

      printf("\ntekst jawny: \"%s\" ", cltext);
}

/****************************************************************************
      FUNKCJA read_rotors wczytuje plik kół kodowych
****************************************************************************/

read_rotors(fp_rotor)
FILE *fp_rotor;
{
     char line[ROTOR_SIZE + 2];
     int i;
     int rotor_number = 0;

     while (fgets(line, ROTOR_SIZE + 2, fp_rotor))
       {
         for (i = 0; i < ROTOR_SIZE; i++)
```

```
            *(rotor[rotor_number].text + i) = line[i];
          rotor_number++;

      }
}

/*************************************************************************
FUNKCJA turn_rotor obraca koło o podanym numerze i ewentualnie sąsiednie
*************************************************************************/

turn_rotor(rotor_number)
int rotor_number;
{
    if ((++(rotor[rotor_number].letter)) == ROTOR_SIZE)
      {
        rotor[rotor_number].letter = 0;
        if (rotor_number < NUM_ROTORS - 1)
          turn_rotor(++rotor_number);
    }
}

/*************************************************************************

               KONIEC PLIKU cipher.c

*************************************************************************/
```

Dodatek D. Tworzenie drzewa binarnego z wyrażenia wielomianowego

Program z tego dodatku czyta wyrażenie takie jak (a+b)*c, a następnie tworzy i pokazuje drzewo binarne reprezentujące to wyrażenie. Nawiasy mogą być dowolnie zagłębione; operator * jest przyjmowany domyślnie tam, gdzie nie ma żadnego operatora. Wszystkie operatory mają ten sam priorytet (odmiennie niż w C).

Drzewo binarne jest drzewem, w którym każdy węzeł ma zero, jeden lub dwa węzły potomne. Każdy węzeł, z wyjątkiem węzła zwanego korzeniem, ma jednego przodka. Można sobie wyobrazić drzewo jak roślinę z korzeniem na czubku i gałęziami skierowanymi w dół. Drzewo binarne jest ważną i użyteczną strukturą danych. Zauważmy, że definicja jest z zasady rekurencyjna, co zresztą prowadzi do użycia rekurencyjnych algorytmów.

Program demonstruje trzy ważne elementy programowania: rekurencję, dynamiczny przydział pamięci i użycie wskaźników.

```
#include <stdio.h>
#include <ctype.h>

#define TRUE    1
#define FALSE   0
#define SUCCESS  0
#define FAILURE    1

#define SPACE ' '
#define EOS '\0'
#define EOL '\n'

#define LINE_SIZE 128

/* operatory */

#define OPEN_PAREN '('            /* nawias otwierający */
#define CLOSE_PAREN ')'           /* nawias zamykający */
#define PLUS '+'
#define MINUS '-'
#define MULTIPLY '*'              /* razy  */
#define DIVIDE '/'                /* podziel */
```

```
/* koniec operatorów */

struct node                    /* węzeł drzewa */
    {
        struct node *parent;           /* przodek */
        struct node *left_child;       /* lewy potomek */
        struct node *right_child;      /* prawy potomek */
        char c;
    };

struct node *current_node = NULL; /* przetwarzany węzeł */
struct node *root_node = NULL;     /* korzeń */

/************************************************************************
            deklaracje funkcji
************************************************************************/

struct node *init_node();
struct node *node_alloc();
struct node *push_tree();
struct node *pop_tree();
struct node *insert_above();

/************************************************************************
            main
************************************************************************/

main()
{
    char line[LINE_SIZE];
    char *line_ptr;
    char *fgets();
    char c;

    while (fgets(line,(LINE_SIZE - 1),stdin))
      {
        line_ptr = line;
        current_node = init_node(root_node = NULL);
        while ((c = *(line_ptr++)))
          if (!isspace(c))
              build_tree(c);
        print_tree(root_node,0);
        free_tree(root_node);
      }
}
/**************************************************************************
FUNKCJA build_tree tworzy drzewo na podstawie wyrażenia wielomianowego
**************************************************************************/

build_tree(c)
char c;
{
    switch (c)
      {
        case OPEN_PAREN:
          if (push_tree() == NULL)
```

```
                printf("\nbłąd nr 110");
            break;

        case CLOSE_PAREN:
            if (pop_tree() == NULL)
                printf("\nbłąd nr 120");
            break;

        case PLUS:
        case MINUS:
        case MULTIPLY:
        case DIVIDE:
            if (current_node->c != SPACE)
                if(insert_above() == NULL)
                    return NULL;

            current_node->c = c;
            break;

        default:
            if (push_tree() == NULL)
                printf("\nbłąd nr 180");
            current_node->c = c;
            if (pop_tree() == NULL)
                printf("\nbłąd nr 190");
            break;
        }
    return SUCCESS;
}

/**************************************************************************
FUNKCJA push_tree zwraca wskazanie do nowego węzła na niższym poziomie
**************************************************************************/

struct node *push_tree()
{
    struct node *new_node = NULL; /* wskaźnik do nowego węzła */

    if (current_node == NULL)
        return NULL;

/* jeżeli próba zejścia na niższy poziom przy pełnym węźle, trzeba wstawić dodatkowy
węzeł powyżej i wtedy zejść */

    if ((current_node->left_child) && (current_node->right_child))
        if (insert_above() == NULL)
            return NULL;

/* wstaw domyślny operator mnożenia tam, gdzie go nie ma */

    if ((current_node->left_child) && (current_node->c == SPACE))
        current_node->c = MULTIPLY;

    if ((new_node = init_node()) == NULL)
        return NULL;

    if (current_node->left_child == NULL)
        current_node->left_child = new_node;
    else
        if (current_node->right_child == NULL)
```

```
            current_node->right_child = new_node;
          else
            return NULL;

      new_node->parent = current_node;

      return (current_node = new_node);
}

/*****************************************************************************
FUNKCJA pop_tree powoduje wejście o jeden poziom wyżej z każdego węzła,
z wyjątkiem korzenia
*****************************************************************************/

struct node *pop_tree()
{
     if (current_node == NULL)
          return NULL;

     if (current_node->parent == NULL)
          return NULL;
     else
          return (current_node = current_node->parent);
}

/*****************************************************************************
FUNKCJA delete_node usuwa wskazany węzeł z drzewa
*****************************************************************************/

delete_node(node)
struct node *node;
{
     if (node == NULL)
          return SUCCESS;

     if (node->parent)
       {
          if (node->parent->left_child == node)
                node->parent->left_child = NULL;
          else
             if (node->parent->right_child == node)
                node->parent->right_child = NULL;
       }
     free((char *)node);

     return SUCCESS;
}

/*****************************************************************************
FUNKCJA insert_above dołącza węzeł między tym wskazywanym przez
current_node a jego przodkiem
*****************************************************************************/

struct node *insert_above()
{
     struct node *new_node;           /* nowy węzeł */

     if ((new_node = init_node()) == NULL)
       {
```

```
        printf("\nbłąd nr 220");
        return NULL;
      }

    new_node->left_child = current_node;
    new_node->parent = current_node->parent;

    if (current_node->parent)
      {
        if (current_node->parent->right_child == current_node)
              current_node->parent->right_child = new_node;
        else
           if (current_node->parent->left_child == current_node)
              current_node->parent->left_child = new_node;
      }
    else
      root_node = new_node;

    current_node->parent = new_node;

    return (current_node = new_node);
}
/*************************************************************************
FUNKCJA init_node zwraca wskazanie do utworzonego i zainicjowanego (lecz
nie dołączonego) węzła
*************************************************************************/

struct node *init_node()
{
    struct node *node = NULL;

    if ((node = node_alloc()) == NULL)
      {
        printf("\nbłąd nr 300");
        return NULL;
      }

    if (root_node == NULL)
      root_node = node;

    node->parent = NULL;
    node->left_child = NULL;
    node->right_child = NULL;
    node->c = SPACE;

    return node;
}
/*************************************************************************
FUNKCJA free_tree zwalnia pamięć zajmowaną przez drzewo
*************************************************************************/

free_tree(node)
struct node *node;
{
    if (node)
      {
        if (node->left_child)
```

```
            free_tree(node->left_child);
        if (node->right_child)
            free_tree(node->right_child);

        delete_node(node);
    }
}

/***************************************************************************
FUNKCJA print_tree pokazuje drzewo – pomocna przy uruchamianiu
***************************************************************************/

print_tree(node,level)
struct node *node;
int level;
{
    int i = level;

    while (i--)
        printf("\t");

    printf("%c\n",node->c);

    if (node->parent)
        {
            if ((node->parent->right_child != node)
                && (node->parent->left_child != node))
                    printf("\tBŁĄD -WĘZEŁ BEZ PRZODKA !!!!");
        }

    if (node->left_child)
            print_tree(node->left_child,level + 1);

    if (node->right_child)
            print_tree(node->right_child,level + 1);
}

/***************************************************************************
FUNKCJA node_alloc przydziela pamięć na nowy węzeł
***************************************************************************/

struct node *node_alloc()
{
    return ((struct node*)malloc(sizeof(struct node)));
}

/***************************************************************************
        KONIEC PROGRAMU
***************************************************************************/
```

Dodatek E. Tablica znaków ASCII

Dziesiętnie	Szesnastkowo	Ósemkowo	Znak	Mnemonik
0	00	000	^@	NUL
1	01	001	^A	SOH
2	02	002	^B	STX
3	03	003	^C	ETX
4	04	004	^D	EOT
5	05	005	^E	ENQ
6	06	006	^F	ACK
7	07	007	^G	BEL
8	08	010	^H	BS
9	09	011	^I	HT
10	0a	012	^J	HT
11	0b	013	^K	VT
12	0c	014	^L	FF
13	0d	015	^M	CR
14	0e	016	^N	SO
15	0f	017	^O	SI
16	10	020	^P	DLE
17	11	021	^Q	DC1
18	12	022	^R	DC2
19	13	023	^S	DC3
20	14	024	^T	DC4
21	15	025	^U	NAK
22	16	026	^V	SYN
23	17	027	^W	ETB
24	18	030	^X	CAN
25	19	031	^Y	EM
26	1a	032	^Z	SUB
27	1b	033	ESCAPE	ESC
28	1c	034		FS
29	1d	035		GS
30	1e	036		RS
31	1f	037		US
32	20	040	SPACJA	
33	21	041	!	
34	22	042	"	
35	23	043	#	
36	24	044	S	

37	25	045	%
38	26	046	&
39	27	047	'
40	28	050	(
41	29	051	)
42	2a	052	*
43	2b	053	+
44	2c	054	,
45	2d	055	-
46	2e	056	.
47	2f	057	/
48	30	060	0
49	31	061	1
50	32	062	2
51	33	063	3
52	34	064	4
53	35	065	5
54	36	066	6
55	37	067	7
56	38	070	8
57	39	071	9
58	3a	072	:
59	3b	073	;
60	3c	074	<
61	3d	075	=
62	3e	076	>
63	3f	077	?
64	40	100	@
65	41	101	A
66	42	102	B
67	43	103	C
68	44	104	D
69	45	105	E
70	46	106	F
71	47	107	G
72	48	110	H
73	49	111	I
74	4a	112	J
75	4b	113	K
76	4c	114	L
77	4d	115	M
78	4e	116	N
79	4f	117	O
80	50	120	P
81	51	121	Q
82	52	122	R
83	53	123	S
84	54	124	T
85	55	125	U
86	56	126	V
87	57	127	W
88	58	130	X
89	59	131	Y
90	5a	132	Z
91	5b	133	[

92	5c	134	\	
93	5d	135	]	
94	5e	136	^	
95	5f	137	_	
96	60	140	`	
97	61	141	a	
98	62	142	b	
99	63	143	c	
100	64	144	d	
101	65	145	e	
102	66	146	f	
103	67	147	g	
104	68	150	h	
105	69	151	i	
106	6a	152	j	
107	6b	153	k	
108	6c	154	l	
109	6d	155	m	
110	6e	156	n	
111	6f	157	o	
112	70	160	p	
113	71	161	q	
114	72	162	r	
115	73	163	s	
116	74	164	t	
117	75	165	u	
118	76	166	v	
119	77	167	w	
120	78	170	x	
121	79	171	y	
122	7a	172	z	
123	7b	173	{	
124	7c	174	|	
125	7d	175	}	
126	7e	176	~	
127	7f	177	DELETE	DEL

Skorowidz

A

abs 106, 124
acos 106, 124
adres zmiennej 36
adresowanie pośrednie 28
agregat 97
Algol 8
American National Standards Institute 8, 79
anachronizm 37
argc 38
argument 39
argv 38
asin 106, 125
atan 106,125
atan2 106, 125
atof 106, 125
atoi 106, 125
auto 13, 59, 77

B

biblioteka funkcji 34, 40
blok 16, 41
błędnie użyte nawiasy 28
błędy 25, 41
– częste w C 28
break 16, 41, 77

C

calloc 106, 129
case 16, 42, 77
ceil 106, 126
char 12, 13, 77, 86
ciąg znaków 13, 42
clearerr 104, 110
clock 107, 130
close 104, 108
continue 16, 43, 77
cos 106, 126
cosh 106, 126
CP/M 80, 8
creat 104, 108
ctime 107, 130
czas życia zmiennej 44

D

default 77, 84
definicja funkcji 44
– zmiennej 44
– danych 12
deklaracja funkcji 46
– zmiennej 47
– danych 12
deszyfrowanie 140
do 16, 77
do .. while 16, 47
double 12, 13, 77, 86
drzewa binarne 146
dyrektywy preprocesora 100

E

efekty uboczne 47
else 77
Enigma 140
entry 77
enum 48, 77, 13

eof 104, 108
EOF 48
etykieta 49
exit 107, 131
_exit 107, 131
exp 106, 126
extern 13, 45, 77, 52, 59

F

fabs 106, 126
fclose 104, 110
feof 104, 110
ferror 104, 110
fflush 104, 110
fgetc 104, 111
fgets 104, 111
fileno 104,111
float 12, 13, 77, 86, 132
floor 106, 126
fmod 106, 126
fopen 104, 111
for 49, 77
fprintf 104, 112
fputc 104, 112
fputs 104, 112
fread 104, 112
free 106, 129
freopen 105, 113
fscanf 105, 113
fseek 105,113
ftell 105, 114
funkcje 18, 32, 33, 51, 104
– buforowane wejscia-wyjscia 104, 110
– daty i czasu 107, 129
– matematyczne 106, 124
– niebuforowane wejścia-wyjścia 104, 108
– obsługi pamięci 106, 128
– przetwarzania ciągów znaków 105, 121
– różne 107, 131
– standardowe 104
– wejścia-wyjścia 104, 108, 110
fwrite 105, 114

G

getc 105, 114
getchar 105,114
gets 105, 115
getw 105, 115
gmtime 107, 130
goto 16, 33, 52, 77

H

Hanoi wieża 75
Hopper Grace Murray 21

I

IBM PC 11
identyfikator zewnętrzny 54
identyfikatory 12, 53
– ciągów znaków 12
– zmiennych całkowitych 12
– zmienych znakowych 12
idiomy 33, 54
if..else 16, 54, 77
index 105, 118
inicjowanie 55
instrukcja pusta 16
instrukcje 16, 58
int 12, 13, 77, 86
isalnum 105, 122
isalpha 105, 122
isascii 105, 122
iscntrl 106, 122
isdigit 106, 122
isgraph 106, 122
islower 106, 123
isprint 106, 123
ispunct 106, 123
isspace 106, 123
isupper 106, 123

K

Kernighan Brian 7
klasa pamięci 59
– – pominięta 79
kod źródłowy 34
kolejność obliczeń 60
komentarze 29, 32, 60
kompilacja 61
konsolidator 62
konwersja typów 62
koszty opracowania i konserwacji programu 31

L

Levi Primo 25
liczba całkowita 17
– ósemkowa 13
– szesnastkowa 13
lint 11
localtime 107, 130

log 106, 127
log10 106, 127
long 12, 13, 77, 86
long int 12, 13, 77, 86
lseek 104, 108
l-wartość 64

Ł

łączność 64, 67

M

main 83
makrodefinicja 65
malloc 106, 129
memccpy 105, 118
memchr 105, 118
memcmp 105, 118
memcpy 105, 118
memset 105, 109
metoda dedukcyjna 23

N

naukowa metoda uruchamiania 23
nawiasy klamrowe 32
niezerowa wartość 85
niezgodność między parametrem formalnym
a aktualnym 30
notacja naukowa 13, 78, 125
NULL 67

O

open 104, 109
operator dwuargumentowy 67
– jednoargumentowy 67
– konwersji 63
– trójargumentowy 67
operatory 15, 67

P

Pascal 8, 28, 55
plik źródłowy 11, 71
pliki 11
– standardowe 71
pola bitowe 71
postać pośrednia 61
pow 106, 127
poziom składniowy 18, 83
preprocesor 19, 72
printf 105, 115
priorytet operatorów 73
program błędnie działający 24
–, koszty konserwacji 31
–, koszty opracowania 31
–, struktura 80
przenośność 73
putc 105, 115
putchar 105, 116
puts 105, 116
putw 105, 116

Q

qsort 107, 132

R

rand 106, 127
random 106, 127
read 104, 109
realloc 106, 129
register 13, 59, 77
rekurencja 75, 146
return 16, 75, 77
rindex 105, 119
Ritchie Dennis 7
rozszerzenie znaku 76

S

słowa kluczowe 77
scanf 105, 116
sekwencje specjalne 14, 76
short 77
short int 12, 13, 77, 86
sin 106, 127
sinh 106, 128
sizeof 15, 67, 77
skomplikowane warunki 32
sprawdzanie typów 77
sprintf 105, 119
sqrt 106, 128
srand 106, 128
sscanf 105, 116
stała 13, 77
stałe symboliczne 79
standard ANSI języka C 8, 79
static 13, 59, 77
stderr 80
stdin 80
stdio.h 48, 80
stdout 80
strcat 105, 119

strchr 105, 120
strcmp 105, 120
strcpy 105, 120
strlen 105, 120
strncat 105, 120
strncmp 105, 121
strncpy 105, 121
strrchr 105, 121
struct 77
struktura 13, 80
styl programowania 31
switch 16, 77, 83
system 107, 131
szyfrowanie 140

Ś

środki pracy konwersacyjnej 22

T

tablica 12, 17, 84
tan 106, 128
tanh 106, 128
tekst sterujący 132, 133
tell 104, 109
time 107, 139
tmpfile 105, 117
toascii 106, 124
tolower 106, 124
toupper 106, 124
tworzenie nazw 12
typedef 77, 86
typy danych 86

U

ungetc 105, 117
unia 13, 87
union 77, 87
Unix 61, 90
unlink 105, 117
unsigned 77
unsigned char 12, 86
unsigned int 12, 86
uruchamianie programów 21
ustawienie 91

V

void 77

W

while 16, 47, 77, 92
wiersz kontynuacji 92
write 104, 109
wskaźnik 13, 17
– do funkcji 93
– do zmiennej 94
–, arytmetyka 40
–, użycie 146
wyrażenie 16, 96
– wielomianowe 140
wywołanie przez nazwę 96
– – wartość 96
zasięg 98
zmienne 12, 36
zmienne globalne 32

Mikrokomputery

W serii ukazały się następujące pozycje:

J. Bielecki – Fortan 77
J. Bielecki – Język C – interpretacja standardu
J. Bielecki – Język Forth
J. Bielecki – System operacyjny ISIS-II
J. Bielecki – Turbo C z grafiką dla IBM PC
J. Bielecki – Turbo Pascal wersja 3.0
J. Bielecki – Turbo Pascal z grafiką dla IBM PC
J. Bielecki – Wprowadzenie do języka C
J Boisgontier, S. Brébion – Basic dla wszytkich
L. Bułhak, R. Goczyński, M. Tuszyński – System operacyjny MS DOS, PC DOS
W. Celary, Z. Królikowski – Wprowadzenie do projektowania baz danych – dBase III
W. Cellary, J. Rykowski – System operacyjny CP/J dla mikrokomputera Elwro 800 Junior
W. Cellary, W. Wieczerzycki – Wielozadaniowy system operacyjny czasu rzeczywistego iRMX-88
B. Frelek – Commodore 64
D. Hearn, M. P. Baker – Grafika mikrokomputerowa
W. Iszkowski – Nauka programowania w języku BASIC dla początkujących
W. Iszkowski – Nauka programowania w języku BASIC dla zaawansowanych
M. Kalinowska-Iszkowska, W. Iszkowski – Klucze do Basicu
J. Karczmarczuk – Mikroprocesor Z80
W. Link – Jak mierzyć, sterować i regulować za pomocą Basicu?
A. Plaza, E. J. Wróbel – Systemy czasu rzeczywistego
A. Ragen – Leksykon języka C
W. Sikorski – Wprowadzenie do użytkowania mikrokomputerów
R. Świniarski – System operacyjny CP/M
A. Wolisz – Podstawy lokalnych sieci komputerowych. Sprzęt sieciowy

W przygotowaniu

M. Adamski, J. Szajna – Programowanie logiczne – Turbo Prolog
J. K. Borowicki – Język RDSQL
W. Cellary, K. Pielesiak – Leksykon Logo
R. Goczyński, M. Tuszyński – Procesory 80286 i 80386 oraz koprocesory arytmetyczne 80287 i 80387 – działanie i programowanie
W. Iszkowski – System OS/2
J. Lansdown – Grafika komputerowa
H. Małysiak, B. Pochopień, E. Wróbel – Mikrokomputery klasy IBM PC
A. Nafalski, M. Wójtowicz – Programowanie strukturalne w języku Turbo Basic
Z. Sołtys – System operacyjny Xenix
S. Waligórski – Programowanie w języku Logo
A. Wolisz – Podstawy lokalnych sieci komputerowych. Oprogramowanie
E. Wróbel – Asembler 8086/8088

WNT Warszawa 1990. Wydanie I. Nakład 9800+200 egz.
Format B5. Ark. wyd. 12,0. Ark. druk. 10,0.
Papier offset. kl. III, 70 g. Podpisano do druku
w styczniu 1990. Druk ukończono w styczniu 1990.
Symbol Et/82532/WNT
PZ „Polmark". Zam. 765/90